Сергей Сабуров

Языки программирования
С и С++

УДК 681.3
ББК 32.973.26-018.2
С418

Сабуров С.В.
С418 Языки программирования С и С++. - М.: Бук-пресс, 2006. - 647 с. - (Справочное руководство пользователя персонального компьютера).

ISBN 5-8321-0138-2

В книге («неофициальное» руководство пользователя) полностью описаны языки программирования С и С++. Уделено особое внимание описанию языка С++ и интегрированных средах разработки программ TURBO C++ и Visual C. Язык программирования С++ — это С, расширенный введением классов, inline-функций, перегруженных операций, перегруженных имен функций, константных типов, ссылок, операций управления свободной памятью, проверки параметров функций. Этот язык, сохранив средства ставшего общепризнанным стандартом для написания системных и прикладных программ языка С (процедурно-ориентированный язык), ввел в практику программирования возможности нового технологического подхода к разработке программного обеспечения, получившего название «объектно-ориентированное программирование».

УДК 681.3
ББК 32.973.26-018.2

ISBN 5-8321-0139-2

Содержание

Язык программирования Си

Язык программирования C++

Отладчик Turbo Debugger

Турбо Си ++

Трюки программирования

Приложения

Язык программирования Си

Введение

Си — это язык программирования общего назначения, хорошо известный своей эффективностью, экономичностью, и переносимостью. Указанные преимущества Си обеспечивают хорошее качество разработки почти любого вида программного продукта. Использование Си в качестве инструментального языка позволяет получать быстрые и компактные программы. Во многих случаях программы, написанные на Си, сравнимы по скорости с программами, написанными на языке ассемблера. При этом они имеют лучшую наглядность и их более просто сопровождать.

Си сочетает эффективность и мощность в относительно малом по размеру языке. Хотя Си не содержит встроенных компонент языка, выполняющих ввод-вывод, распределение памяти, манипуляций с экраном или управление процессами, тем не менее, системное окружение Си располагает библиотекой объектных модулей, в которой реализованы подобные функции. Библиотека поддерживает многие из функций, которые требуются.

Это решение позволяет изолировать языковые особенности от специфики процессора, на котором выполняется результирующая программа. Строгое определение языка делает его независимым от любых деталей операционной системы или машины. В то же время программисты могут добавить в библиотеку специфические системные программы, чтобы более эффективно использовать конкретной особенности машины.

Перечислим некоторые существенные особенности языка Си:

- Си обеспечивает полный набор операторов структурного программирования.
- Си предлагает необычно большой набор операций. Многие операции Си соответствуют машинным командам и поэтому допускают прямую трансляцию

в машинный код. Разнообразие операций позволяет выбирать их различные наборы для минимизации результирующего кода.

- Си поддерживает указатели на переменные и функции. Указатель на объект программы соответствует машинному адресу этого объекта.

Посредством разумного использования указателей можно создавать эффективно-выполняемые программы, так как указатели позволяют ссылаться на объекты тем же самым путем, как это делает машина. Си поддерживает арифметику указателей, и тем самым позволяет осуществлять непосредственный доступ и манипуляции с адресами памяти.

В своем составе Си содержит препроцессор, который обрабатывает текстовые файлы перед компиляцией. Среди его наиболее полезных приложений при написании программ на Си являются: определение программных констант, замена вызовов функций аналогичными, но более быстрыми макросами, условная компиляция. Препроцессор не ограничен процессированием только исходных текстовых файлов Си, он может быть использован для любого текстового файла.

Си — гибкий язык, позволяющий принимать в конкретных ситуациях самые разные решения. Тем не менее, Си налагает незначительные ограничения в таких, например, действиях, как преобразование типов. Во многих случаях это является достоинством, однако программисты должны хорошо знать язык, чтобы понимать, как будут выполняться их программы.

Множества символов

В программах на Си используется два множества символов: множество символов Си и множество представимых символов. Множество символов Си содержит буквы, цифры и знаки пунктуации, которые имеют определенное значение для компилятора Си. Программы на Си строятся путем комбинирования в осмысленные конструкции символов из множества Си.

Множество символов Си является подмножеством множества представимых символов. Множество представимых символов состоит из всех букв, цифр и символов, которые

пользователь может представить графически как отдельный символ. Мощность множества представимых символов зависит от типа терминала, который используется.

Программа на Си может содержать только символы из множества символов Си, за исключением строковых литералов, символьных констант и комментариев, где может быть использован любой представимый символ. Каждый символ из множества символов Си имеет вполне определенный смысл для компилятора Си. Компилятор выдает сообщение об ошибке, когда он встречает неверно использованные символы или символы, не принадлежащие множеству Си.

Буквы и цифры

Множество символов Си включает большие и малые буквы из английского алфавита и 10 десятичных арабских цифр:

- большие английские буквы:

`A B C D E F G H I J K L M N O P Q R T U V W X Y Z`

- малые английские буквы:

`a b c d e f g h i j k l m n o p q r t u v w x y z`

- десятичные цифры:

`0 1 2 3 4 5 6 7 8 9`

Буквы и цифры используются при формировании констант, идентификаторов и ключевых слов. Все эти конструкции описаны ниже.

Компилятор Си рассматривает одну и ту же малую и большую буквы как отличные символы. Если в данной записи использованы малые буквы, то замена малой буквы **a** на большую букву **A** сделает отличной данную запись от предшествующей.

Пробельные символы

Пробел, табуляция, перевод строки, возврат каретки, новая страница, вертикальная табуляция и новая строка — это символы, называемые пробельными, поскольку они имеют то же самое назначение, как и пробелы между словами и строками на печатной странице. Эти символы разделяют объекты, определенные пользователем, такие, как константы и идентификаторы, от других объектов программы.

Символ **CONTROL-Z** рассматривается как индикатор конца файла. Компилятор игнорирует любой текст, следующий за символом **CONTROL-Z**.

Компилятор Си игнорирует пробельные символы, если они не используются как разделители или как компоненты константы-символа или строковых литералов. Это нужно иметь в виду, чтобы дополнительно использовать пробельные символы для повышения наглядности программы (например, для просмотра редактором текстов).

Знаки пунктуации и специальные символы

Знаки пунктуации и специальные символы из множества символов Си используются для различных целей, от организации текста программы до определения заданий, которые будут выполнены компилятором или откомпилированной программой.

Эти символы имеют специальный смысл для компилятора Си. Их использование в языке Си описывается в дальнейшем содержании руководства. Знаки пунктуации из множества представимых символов, которые не представлены в данном списке, могут быть использованы только в строковых литералах, константах-символах и комментариях.

ESC-последовательности

ESC-последовательности — это специальные символьные комбинации, которые представляют пробельные символы и не графические символы в строках и символьных константах.

Их типичное использование связано со спецификацией таких действий, как возврат каретки и табуляция, а также для задания литеральных представлений символов, таких как символ двойная кавычка. ESC-последовательность состоит из наклонной черты влево, за которой следует буква, знаки пунктуации, «**'**», «**"**», «****» или комбинация цифр.

Если наклонная черта влево предшествует символу, не включенному в этот список, то наклонная черта влево игнорируется, а символ представляется как литеральный. Например, изображение **\c** представляет символ "c" в литеральной строке или константе-символе.

Последовательности **\ddd** и **\xdd** позволяют задать любой символ в ASCII (Американский стандартный код

информационного интерфейса) как последовательность трех восьмеричных цифр или двух шестнадцатеричных цифр. Например, символ пробела может быть задан как **\010** или **\x08**. Код ASCII **нуль** может быть задан как **\0** или **\x0**. В восьмеричной ESC-последовательности могут быть использованы от одной до трех восьмеричных цифр. Например, символ пробела может быть задан как **\10**. Точно так же в шестнадцатеричной ESC-последовательности могут быть использованы от одной до двух шестнадцатеричных цифр. Так, шестнадцатеричная последовательность для символа пробела может быть задана как **\x08** или **\x8**.

Важно: Когда используется восьмеричная или шестнадцатеричная ESC-последовательность в строках, то нужно полностью задавать все цифры ESC-последовательности (три цифры для восьмеричной и две цифры для шестнадцатеричной ESC- последовательностей).

Иначе, если символ непосредственно следующий за ESC-последовательностью, случайно окажется восьмеричной или шестнадцатеричной цифрой, то он проинтерпретируется как часть последовательности. Например, строка **\x7Bell** при выводе на печать будет выглядеть как **ell**, поскольку **\x7B** проинтерпретируется как символ левой фигурной скобки **()**. Строка **\x07Bell** будет правильным представлением символа **звонок** с последующим словом **Bell**.

ESC-последовательности позволяют посылать неграфические управляющие символы к внешним устройствам. Например, ESC-последовательность **\033** часто используется как первый символ команд управления терминалом и принтером. Неграфические символы всегда должны представляться ESC-последовательностями, поскольку, непосредственное использование в программах на Си неграфических символов будет иметь непредсказуемый результат.

Наклонная черта влево \ помимо определения ESC-последовательностей используется также, как символ продолжения строки в препроцессорных определениях.

Если символ **новая строка** следует за наклонной чертой влево, то новая строка игнорируется и следующая строка рассматривается, как часть предыдущей строки.

Операции

Операции — это специальные комбинации символов, специфицирующие действия по преобразованию различных величин. Компилятор интерпретирует каждую из этих комбинаций как самостоятельную единицу, называемую лексемой **token**.

Ниже представлен список операций. Операции должны использоваться точно так, как они тут представлены: без пробельных символов между символами в тех операциях, которые представлены несколькими символами.

Операция **sizeof** не включена в этот список, так как она представляет собой ключевое слово, а не символ.

!

Логическое НЕ

~

Побитовое дополнение

+

Сложение

-

Вычитание, арифметическое отрицание

*****

Умножение

/

Деление

%

Остаток

<<

Сдвиг влево

>>

Сдвиг вправо

<

Меньше

<=

Меньше или равно

>
Больше

>=
Больше или равно

==
Равно

!=
Не равно

&
Побитовое И, адрес от

|
Побитовое включающее ИЛИ

^
Побитовое исключающее ИЛИ

&&
Логическое И

||
Логическое ИЛИ

,
Последовательное выполнение (запятая)

?:
Операция условного выражения

++
Инкремент

--
Декремент

=
Простое присваивание

+=
Сложение с присваиванием

-=
Вычитание с присваиванием

***=**

Умножение с присваиванием

/=

Деление с присваиванием

%=

Остаток с присваиванием

>>=

Сдвиг вправо с присваиванием

<<=

Сдвиг влево с присваиванием

&=

Побитовое И с присваиванием

|=

Побитовое включающее ИЛИ с присваиванием

^=

Побитовое исключающее ИЛИ с присваиванием

Важно: Операция условного выражения **?:** является тернарной, а не двухсимвольной операцией. Формат условного выражения следующий:

```
<expression>?<expression>:<expression>
```

Константы

Константа — это число, символ или строка символов. Константы используются в программе как неизменяемые величины. В языке Си различают четыре типа констант: целые константы, константы с плавающей точкой, константы-символы и строчные литералы.

Целые константы

Целая константа — это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целую величину. Десятичная константа имеет следующий формат представления:

```
<digits>
```

где **<digits>** — это одна или более десятичных цифр от 0 до 9.

Восьмеричная константа имеет следующий формат представления:

```
0<odigits>
```

где **<odigits>** — это одна или более восьмеричных цифр от 0 до 7. Запись ведущего нуля необходима.

Шестнадцатеричная константа имеет один из следующих форматов представления:

```
0x<hdigits>
0X<hdigits>
```

где **<hdigits>** одна или более шестнадцатеричных цифр.

Шестнадцатеричная цифра может быть цифрой от 0 до 9 или буквой (большой или малой) от A до F. В представлении константы допускается «смесь» больших и малых букв. Запись ведущего нуля и следующего за ним символа x или X необходима.

Пробельные символы не допускаются между цифрами целой константы. Ниже иллюстрируются примеры целых констант.

```
10        012           0xa или 0xA
132       0204          0x84
32179     076663        0x7dB3 или 0x7DB3
```

Целые константы всегда специфицируют положительные величины. Если требуется отрицательные величины, то необходимо сформировать константное выражение из знака минус и следующей за ним константы. Знак минус рассматривается как арифметическая операция.

Каждая целая константа специфицируется типом, определяющим ее представление в памяти и область значений. Десятичные константы могут быть типа **int** или **long**.

Восьмеричные и шестнадцатеричные константы в зависимости от размера могут быть типа **int**, **unsigned int**, **long** или **unsigned long**. Если константа может быть представлена как **int**, она специфицируется типом **int**. Если ее величина больше, чем максимальная положительная величина, которая может быть представлена типом **int**, но меньше величины, которая представляется в том же самом числе бит как и **int**, она задается типом **unsigned int**. Наконец, константа, величина которой больше чем максимальная величина, представляемая типом **unsigned int**, задается типом **long** или **unsigned long**, если это необходимо.

Важность рассмотренных выше правил состоит в том, что восьмеричные и шестнадцатеричные константы не содержат «знаковых» расширений, когда они преобразуются к более длинным типам.

Программист может определить для любой целой константы тип **long**, приписав букву **l** или **L** в конец константы.

Константы с плавающей точкой

Константа с плавающей точкой — это действительное десятичное положительное число. Величина действительного числа включает целую, дробную части и экспоненту. Константы с плавающей точкой имеют следующий формат представления:

```
[<digits>][.<digits>][E[-]<digits>]
```

где **<digits>** — одна или более десятичных цифр (от 0 до 9), а **E** или **e** — символ экспоненты. Целая или дробная части константы могут быть опушены, но не обе сразу. Десятичная точка может быть опущена только тогда, когда задана экспонента.

Экспонента состоит из символа экспоненты, за которым следует целочисленная величина экспоненты, возможно отрицательная.

Пробельные символы не могут разделять цифры или символы константы.

Константы с плавающей точкой всегда специфицируют положительные величины. Если требуются отрицательные величины, то необходимо сформировать константное выражение из знака минус и следующей за ним константы. Знак минус рассматривается как арифметическая операция.

Примеры констант с плавающей точкой и константных выражений:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25e-4
```

Целая часть константы с плавающей точкой может быть опущена, например:

```
.75
```

```
.0075e2
-.125
-.175E-2
```

Все константы с плавающей точкой имеют тип **double**.

Константа-символ

Константа-символ — это буква, цифра, знак пунктуации или ESC-символ, заключенные в одиночные кавычки. Величина константы-символа равна значению представляющего ее кода символа.

Константа-символ имеет следующую форму представления:

```
"<char>"
```

где **<char>** может быть любым символом из множества представимых символов, включая любой ESC-символ, исключая одиночную кавычку ('), наклонную черту влево (\) и символ новой строки.

Чтобы использовать одиночную кавычку или наклонную черту влево в качестве константы-символа, необходимо вставить перед этими знаками наклонную черту влево. Чтобы представить символ новой строки, необходимо использовать запись **\n**.

Строковые литералы

Строковый литерал — это последовательность букв, цифр и символов, заключенная в двойные кавычки. Строковый литерал рассматривается как массив символов, каждый элемент которого представляет отдельный символ. Строковый литерал имеет следующую форму представления:

```
"<characters>"
```

где **<characters>** — это нуль или более символов из множества представимых символов, исключая двойную кавычку, наклонную черту влево и символ новой строки. Чтобы использовать символ новой строки в строковом литерале, необходимо напечатать наклонную черту влево, а затем символ новой строки.

Наклонная черта влево вместе с символом новой строки будут проигнорированы компилятором, что позволяет формировать строковые литералы, располагаемые более чем в одной строке.

Например, строковый литерал:

```
"Long strings can be bro\
cken into two pieces."
```

идентичен строке:

```
Long strings can be brocken into two pieces.
```

Чтобы использовать двойные кавычки или наклонную черту влево внутри строкового литерала, нужно представить их с предшествующей наклонной чертой влево, как показано в следующем примере:

```
"This is a string literal"
"First \\ Second"
"\"Yes, I do,\" she said."
"The following line shows a null string:"
""
```

Заметим, что ESC-символы (такие как \\ и \") могут появляться в строковых литералах. Каждый ESC-символ считается одним отдельным символом.

Символы строки запоминаются в отдельных байтах памяти. Символ **null** или **\0** является отметкой конца строки. Каждая строка в программе рассматривается как отдельный объект. Если в программе содержатся две идентичные строки, то каждая из них будет храниться в отдельном месте памяти.

Строчные литералы имеют тип **char[]**. Под этим подразумевается, что строка — это массив, элементы которого имеют тип **char**. Число элементов в массиве равно числу символов в строчном литерале плюс один, поскольку символ **null** (отметка конца строки) тоже считается элементом массива.

Идентификаторы

Идентификаторы — это имена переменных, функций и меток, используемых в программе. Идентификатор создается объявлением соответствующей ему переменной или функции. После этого его можно использовать в последующих операторах программы. Идентификатор — это последовательность из одной или более букв, цифр или подчерков (_), которая начинается с буквы или подчерка. Допускается любое число символов в идентификаторе, однако только первые 31 символ распознаются компилятором. (Программы, использующие результат работы

компилятора, такие как, линкер, могут распознавать меньшее число символов).

При использовании подчерков в идентификаторе нужно быть осторожным, поскольку идентификаторы, начинающиеся с подчерка могут совпадать (войти в конфликт) с именами «скрытых» системных программ.

Примеры идентификаторов:

```
temp1
toofpage
skip12
```

Компилятор Си рассматривает буквы верхнего и нижнего регистров как различные символы. Поэтому можно создать отдельные независимые идентификаторы, которые совпадают орфографически, но различаются большими и малыми буквами. Например, каждый из следующих идентификаторов является уникальным:

```
add
ADD
Add aDD
```

Компилятор Си не допускает идентификаторов, которые имеют ту же самую орфографию, что и ключевые слова.

Важно: По сравнению с компилятором, сборщик может в большей степени ограничивать количество и тип символов для глобальных идентификаторов, и в отличие от компилятора не делать различия между большими и малыми буквами.

Ключевые слова

Ключевые слова — это предопределенные идентификаторы, которые имеют специальное значение для компилятора Си. Их можно использовать только так как они определены. Имена объектов программы не могут совпадать с названиями ключевых слов.

Список ключевых слов:

```
auto     double     int        struct
break    else       long       switch
case     enum       register   typedef
char     extern     return     union
```

```
const     float     short     unsigned
continue  for       signed    void
default   goto      sizeof    while
do        if        static    volatile
```

Ключевые слова не могут быть переопределены. Тем не менее, они могут быть названы другим текстом, но тогда перед компиляцией они должны быть заменены посредством препроцессора на соответствующие ключевые слова.

Ключевые слова **const** и **volatile** зарезервированы для будущего использования.

Следующие идентификаторы могут быть ключевыми словами для некоторых приложений:

```
cdecl
far
fortran
huge
near
pascal
```

Комментарии

Комментарий — это последовательность символов, которая воспринимается компилятором как отдельный пробельный символ или, другими словами, игнорируется.

Комментарий имеет следующую форму представления:

```
/*<characters>*/
```

где **<characters>** может быть любой комбинацией символов из множества представимых символов, включая символы новой строки, но исключая комбинацию */.

Это означает, что комментарии могут занимать более одной строки, но не могут быть вложенными.

Комментарии допускаются везде, где разрешены пробельные символы. Компилятор игнорирует символы комментария, в частности, в комментариях допускается запись ключевых слов и это не приведет к ошибке. Так как компилятор рассматривает комментарий как символ пробела, то комментарии не могут появляться внутри лексем.

Следующие примеры иллюстрируют некоторые комментарии:

```
/* Comments can separate and document
lines of a program. */
/* Comments can contain keywords such as for
and while */
/*******************************************
Comments can occupy several lines.
*******************************************/
```

Так как комментарии не могут содержать вложенных комментариев, то следующий пример будет ошибочным:

```
/* You cannot/* nest */ comments */
```

Компилятор распознает первую комбинацию */ после слова nest как конец комментария. Затем, компилятор попытается обрабатывать оставшийся текст и выработает сообщение об ошибке. Чтобы обойти компиляцию комментариев больших размеров, нужно использовать директиву **#if** препроцессора.

Лексемы

Когда компилятор обрабатывает программу, он разбивает программу на группы символов, называемых лексемами. Лексема — это единица текста программы, которая имеет определенный смысл для компилятора и которая не может быть разбита в дальнейшем. Операции, константы, идентификаторы и ключевые слова, описанные в этом разделе, являются примерами лексем. Знаки пунктуации, такие как квадратные скобки, фигурные скобки, угловые скобки, круглые скобки и запятые, также являются лексемами. Границы лексем определяются пробельными символами и другими лексемами, такими как операции и знаки пунктуации. Чтобы предупредить неправильную работу компилятора, запрещаются пробельные символы между символами идентификаторов, операциями, состоящими из нескольких символов и символами ключевых слов.

Когда компилятор выделяет отдельную лексему, он последовательно объединяет столько символов, сколько возможно, прежде чем перейти к обработке следующей лексемы. Поэтому лексемы, не разделенные пробельными символами, могут быть проинтерпретированы неверно.

Например, рассмотрим следующее выражение:

```
i+++j
```

В этом примере компилятор вначале создает из трех знаков плюс самую длинную из возможных операций **++**, а затем обработает оставшийся знак +, как операцию сложения **+**. Выражение проинтерпретируется как **(i++)+(j)**, а не как **(i)+(++j)**. В таких случаях необходимо использовать пробельные символы или круглые скобки, чтобы однозначно определить ситуацию.

Исходная программа

Исходная программа — это совокупность следующих объектов: директив, указаний компилятору, объявлений и определений. Директивы задают действия препроцессора по преобразованию текста программы перед компиляцией. Указания компилятору — это команды, выполняемые компилятором во время процесса компиляции. Объявления задают имена и атрибуты переменных, функций и типов, используемых в программе. Определения — это объявления, определяющие переменные и функции.

Определение переменной в дополнении к ее имени и типу задает начальное значение объявленной переменной. Кроме того, определение предполагает распределение памяти для переменной.

Определение функции специфицирует ее структуру, которая представляет собой смесь из объявлений и операторов, которые образуют саму функцию. Определение функции также задает имя функции, ее формальные параметры и тип возвращаемой величины.

Исходная программа может содержать любое число директив, указаний компилятору, объявлений и определений. Любой из объектов программы имеет определенный синтаксис, описанный в этом руководстве, и каждая составляющая может появляться в любом порядке, хотя влияние порядка, в котором следуют переменные и функции может быть использовано в программе.

Нетривиальная программа всегда содержит более одного определения функции. Функция определяет действия, выполняемые программой.

В следующем примере иллюстрируется простая исходная программа на языке Си.

```
int x = 1;      /* Variable definitions */
int y = 2;
extern int printf(char *,...);/* Function declaration */
main ()  /* Function definition
for main function */
int z;   /* Variable declarations */
int w;
z = y + x;      /* Executable statements */
w = y - x;
printf("z = %d \nw = %d \n", z, x);
```

Эта исходная программа определяет функцию с именем **main** и объявляет функцию **printf**. Переменные **x** и **y** задаются своими определениями. Переменные **z** и **w** только объявляются.

Исходные файлы

Исходные программы могут быть разделены на несколько файлов. Исходный файл Си — это текстовый файл, который содержит часть или всю исходную программу. Он может, например, содержать только некоторые функции, требуемые программе. При компиляции исходной программы каждый из исходных файлов должен быть прокомпилирован отдельно, а затем обработан сборщиком. Отдельные исходные файлы перед компиляцией можно соединять в один большой исходный файл посредством директивы **#include**.

Исходный файл может содержать любую комбинацию наборов: директив, указаний компилятору, объявлений и определений. Такие объекты, как определения функций или большие структуры данных, не могут разрываться, начинаясь в одном файле и продолжаясь в другом.

Исходный файл не обязательно должен содержать выполняемые операторы. Иногда полезно размещать описания переменных в одном файле с тем, чтобы использовать их путем объявления ссылок из других файлов. В этом случае определения становятся легко доступными для поиска и модификации. Из тех же самых соображений константы и макросы часто организуют в отдельных **#include** — файлах и включают их, если требуется, в исходные файлы.

Директивы исходного файла относятся только к этому исходному файлу и файлам, включающим его (посредством **#include**). Кроме того, каждая директива относится только к части файла, которая следует за ней. Если множество директив должно относиться ко всей исходной программе, то все исходные файлы должны содержать эти директивы.

Указания компилятору обычно эффективны для отдельных областей исходного файла. Специфические действия компилятора, задаваемые указаниями, определяются содержанием последних.

В нижеследующем примере исходная программа состоит из двух исходных файлов. Функции **main** и **max** представлены в отдельных файлах. Функция **main** использует функцию max при своем выполнении.

```
/**********************************************
Sourse file 1 - main function
**********************************************/
#define ONE     1
#define TWO     2
#define THREE   3
extern int max(int, int);    /* Function declaration */
main ()   /* Function definition */
int w = ONE, x = TWO, y = THREE;
int z = 0;
z = max(x,y);
w = max(z,w);
/************************************************
Sourse file 2 - max function
************************************************/
int max(a,b)    /* Function definition */
int a, b;
if ( a > b )
return (a);
else
return (b);
```

В первом исходном файле функция max объявлена без ее определения. Это объявление известно как **forward**-объявление. Определение функции main включает вызов функции **max**.

Строки, начинающиеся с символа номер #, являются директивами препроцессора. Директивы инструктируют препроцессор о замене в первом исходном файле имен **ONE**, **TWO**, **THREE** на соответствующие им значения. Область действия директив не распространяется на второй исходный файл.

Второй исходный файл содержит описание функции max. Это определение соответствует вызовам max из первого исходного файла.

Выполнение программ

Каждая программа содержит главную программную функцию. В Си главная программная функция должна быть поименована как **main**. Функция **main** служит точкой старта при выполнении программы и обычно управляет выполнением программы, организуя вызовы других функций. Программа обычно завершает выполнение по окончанию функции **main**, хотя она может завершиться и в других точках, в зависимости от окружающей обстановки.

Исходная программа обычно включает в себя несколько функций, каждая из которых предназначена для выполнения определенного задания. Функция **main** может вызывать эти функции с тем, чтобы выполнить то или иное задание. Функция возвращает управление при выполнении оператора **return** или по окончанию самой функции (выход на конец функции).

Все функции, включая функцию **main**, могут быть объявлены с параметрами. Вызываемые функции получают значения параметров из вызывающих функций. Значения параметров функции **main** могут быть переданы из внешнего окружения.

Например, они могут быть переданы из командной строки.

Соглашение Си требует, чтобы первые два параметра функции **main** назывались **argc** и **argv**.

Параметр **argc** определяет общее число аргументов, передаваемых функции **main**. Параметр **argv** объявляется как массив указателей, каждый элемент которого ссылается на строковое представление аргумента, передаваемого функции **main**. Третий параметр функции **main** (если он есть) традиционно

задается с именем **envp**. Однако Си не требует этого имени. Параметр **envp** — это указатель на массив указателей строковых величин, которые определяют окружение, в котором выполняется программа.

Операционная система поддерживает передачу значений для **argc**, **argv** и **envp** параметров, а пользователь поддерживает задание значений фактических параметров для функции **main**. Соглашение о передаче параметров в большей степени определяется операционной системой, чем самим языком Си.

Формальные параметры функции должны быть объявлены во время описания функции.

«Время жизни» и «Видимость»

Концепции «Время жизни» и «Видимость» являются очень важными для понимания структуры программ на Си. Время жизни переменной может быть или глобальным или локальным. Объект с глобальным временем жизни характеризуется определенной памятью и значением на протяжении всей жизни программы. Объект с локальным временем жизни захватывает новую память при каждом входе в блок, в котором он определен или объявлен. Когда блок завершается, локальный объект пропадает, а следовательно пропадает его значение. Определение блоков описывается ниже.

Объект считается видимым в блоке или исходном файле, если известны тип и имя объекта в блоке или исходном файле. Объект может быть **глобально видимым**, когда имеется в виду, что он видим или может быть объявлен видимым на протяжении всего исходного файла, образующего программу.

Блок — это составной оператор. Составные операторы состоят из объявлений и операторов.

Структура функции представляет собой совокупность составных операторов. Таким образом, функции имеют блочную структуру, блоки, в свою очередь, могут содержать внутри себя другие блоки.

Объявления и определения внутри блоков находятся на **внутреннем уровне**. Объявления и определения вне блоков находятся на **внешнем уровне**. Переменные и функции могут быть объявлены как на внешнем уровне, так и на внутреннем.

Переменные также могут быть определены на внутреннем и на внешнем уровне, а функции определяются только на внешнем уровне.

Все функции имеют глобальное время жизни, невзирая на то, где они объявлены. Переменные, объявленные на внешнем уровне, всегда имеют глобальное время жизни. Переменные, объявленные на внутреннем уровне, всегда имеют локальное время жизни, однако, классы памяти, специфицированные как **static** и **extern**, могут быть использованы для объявления глобальных переменных или ссылок на них внутри блока.

Переменные, объявленные или определенные на внешнем уровне, видимы из точки, в которой они объявлялись или определялись, до конца файла.

Однако, переменные, заданные классом памяти **static** на внешнем уровне, видимы только внутри исходного файла, в котором они определены.

В общем случае, переменные, объявленные или определенные на внутреннем уровне, видимы от точки, в которой они объявлены или определены, до конца блока, в котором представлены объявления или определения. Эти переменные называются локальными. Если переменная, объявленная внутри блока, имеет то же самое имя, как и переменная, объявленная на внешнем уровне, то определение переменной в блоке заменяет (имеет предпочтение) определение внешнего уровня на протяжении блока. Видимость переменной внешнего уровня восстанавливается при завершении блока.

Блочная видимость может вкладываться. Это значит, что блок, вложенный внутрь другого блока, может содержать объявления, которые переопределяют переменные, объявленные во внешнем блоке. Переопределение переменной имеет силу во внутреннем блоке, но первоначальное определение восстанавливается, когда управление возвращается внешнему блоку. Переменные из внешних блоков видимы внутри всех внутренних блоков до тех пор, пока они не переопределены во внутреннем блоке.

Функции класса памяти **static** видимы только в исходном файле, в котором они определены. Все другие функции являются глобально видимыми.

В следующем примере программы иллюстрируются блоки, вложения и видимость переменных.

```
/* i defined at external level */
int i = 1;
/* main function defined at external level */
main ()

/* prints 1 (value of external level i) */
printf("%d\n", i);
/* first nested block */

/* i and j defined at internal level */
int i = 2, j = 3;
/* prints 2, 3 */
printf("%d\n%d\n", i, j);
/* second nested block */

/* i is redefined */
int i = 0;
/* prints 0, 3 */
printf("%d\n%d\n", i, j);
/* end of second nested block */

/* prints 2 (outer definition restored) */
printf("%d\n", i);
/* end of first nested block */

/* prints 1 (external level definition restored */
printf("%d\n", i);
```

В этом примере показано четыре уровня видимости: самый внешний уровень и три уровня, образованных блоками. Предполагается, что функция **printf** определена где-нибудь в другом месте программы. Функция **main** печатает значения 1, 2, 3, 0, 3, 2, 1.

Классы имен

В любой программе на Си имена используются для ссылок на различного рода объекты. Когда пишется программа на Си, то в ней используются идентификаторы для именования функций,

переменных, формальных параметров, областей памяти и других объектов программы. По определенным правилам в Си допускаются использование одного и того же идентификатора для более чем одного программного объекта.

Чтобы различать идентификаторы для различного рода объектов, компилятор устанавливает так называемые **Классы имен**. Для избегания противоречий, имена внутри одного класса должны быть уникальными, однако в различных классах могут появляться идентичные имена. Это означает, что можно использовать один и тот же идентификатор для двух или более различных объектов, если объекты принадлежат к различным классам имен. Однозначное разрешение ссылок компилятор осуществляет по контексту данного идентификатора в программе. Ниже описываются виды объектов, которые можно именовать в программе на Си, а также правила их именования.

Имена переменных и функций относятся к одному классу имен вместе с формальными параметрами и перечислимыми константами. Поэтому, имена переменных и функций этого класса должны быть отличны от других имен с той же видимостью.

Однако, имена переменных могут быть переопределены внутри программных блоков.

Имена функций также могу быть переопределены в соответствии с правилами видимости.

Имена формальных параметров функции появляются вместе с именами переменных функции, поэтому имена формальных параметров должны быть отличны от имен переменных. Переобъявление формальных параметров внутри функции приводит к ошибке.

Перечислимые константы относятся к тому же классу имен, что и имена переменных и функций. Это означает, что имена перечислимых констант должны быть отличны от имен всех переменных и функций с той же самой видимостью. Однако, подобно именам переменных, имена перечислимых констант имеют вложенную видимость, а это означает, что они могут быть переопределены внутри блоков.

Имена **typedef** относятся к одному классу имен вместе с именами переменных и функций. Поэтому они должны быть

отличны от всех имен переменных и функций с той же самой видимостью, а также от имен формальных параметров и перечислимых констант. Подобно именам переменных, имена, используемые для **typedef** типов могут быть переопределены внутри блоков программы.

Теги перечислений, структур и совмещений сгруппированы вместе в одном классе имен. Каждый тег перечисления, структуры или соединения должен быть отличен от других тегов с той же самой видимостью. Теги не конфликтуют с любыми другими именами.

Элементы каждой структуры или совмещения образуют класс имен, поэтому имя элемента должно быть уникальным внутри структуры или совмещения, но это не означает, что они должны быть отличными от любого другого имени в программе, включая имена элементов других структур и совмещений.

Метки операторов формируют отдельный класс имен. Каждая метка оператора должна быть отлична от всех других меток операторов в той же самой функции. Метки операторов могут совпадать с именами меток других функций.

Пример:

```
struct student
char student[20];
int class;
int id;
 student;
```

В этом примере имя тега структуры, элемента структуры и переменной относятся к трем различным классам имен, поэтому не будет противоречия между тремя объектами с именем **student**. Компилятор определит по контексту программы как интерпретировать каждый из случаев. Например, когда имя **student** появится после ключевого слова **struct**, это будет означать, что появился **tag** структуры. Когда имя **student** появится после операции выбора элемента **->** или **.**, то это означает, что имя ссылается на элемент структуры. В другом контексте имя **student** является переменной структурного типа.

Объявления

Объявления Си имеют следующий синтаксис:

```
[<sc-specifier>][<type-specifier>]<declarator>[=<initializer>]
[,<declarator>[=<initializer>...],
```

где:

<sc-specifier>

Спецификатор класса памяти.

<type-specifier>

Имя определяемого типа.

<declarator>

Идентификатор, который может быть модифицирован при объявлении указателя, массива или функции.

<initializer>

Задает значение или последовательность значений, присваиваемых переменной при объявлении.

Все переменные Си должны быть явно объявлены перед их использованием. Функции Си могут быть объявлены явно или неявно в случае их вызова перед определением.

Язык Си определяет стандартное множество типов данных. К этому множеству можно добавлять новые типы данных посредством их объявлений на типах данных уже определенных.

Объявление Си требует одного или более деклараторов. Декларатор — это идентификатор, который может быть определен с квадратными скобками **[]**, звездочкой ***** или круглыми скобками **()** для объявления массива, указателя или функции. Когда объявляется простая переменная (такая как символ, целое или плавающее), структура или совмещение простых переменных, то декларатор — это идентификатор.

В Си определено четыре спецификатора класса памяти, а именно: **auto**, **extern**, **register** и **static**.

Спецификатор класса памяти определяет, каким образом объявляемый объект запоминается и инициализируется и из каких частей программы можно ссылаться на него. Расположение объявления внутри программы, а также наличие или отсутствие других объявлений — также важные факторы при определении видимости переменных.

Спецификаторы типов

Язык Си поддерживает определения для множества базовых типов данных, называемых «основными» типами.

```
signed char     float  void
signed int      double
signed short int
signed long int
unsigned char
unsigned int
unsignet short int
unsigned long int
```

Перечислимые типы также рассматриваются как основные типы.

Типы **signed char**, **signed int**, **signed short int** и **signed long int** вместе с соответствующими двойниками **unsigned** называются типами целых.

Спецификаторы типов **float** и **double** относятся к типу плавающих. В объявлениях переменных и функций можно использовать любые спецификаторы целый и плавающий.

Тип **void** может быть использован только для объявления функций, которые не возвращают значения.

Можно задать дополнительные спецификаторы типа путем объявления **typedef**.

При записи спецификаторов типов допустимы сокращения. В целых типах ключевое слово **signed** может быть опущено. Так, если ключевое слово **unsigned** опускается в записи спецификатора типа, то тип целого будет знаковым, даже если опущено ключевое слово **signed**.

В некоторых реализациях могут быть использованы опции компилятора, позволяющие изменить умолчание для типа **char** со знакового на беззнаковый. Когда задана такая опция, сокращение **char** имеет то же самое значение, что и **unsigned char**, и следовательно ключевое слово **sidned** должно быть записано при объявлении символьной величины со знаком.

Тип **char** используется для запоминания буквы, цифры или символа из множества представимых символов. Значением объекта типа **char** является ASCII код, соответствующий данному

символу. Так как тип **char** интерпретируется как однобайтовая целая величина с областью значений от -128 до 127, то только величины от 0 до 127 имеют символьные эквиваленты. Аналогично, тип **unsigned char** может запоминать величины с областью значений от 0 до 255.

Заметим, что представление в памяти и область значений для типов **int** и **unsigned int** не определены в языке Си. По умолчанию размер **int** (со знаком и без знака) соответствует реальному размеру целого на данной машине. Например, на 16-ти разрядной машине тип **int** всегда 16 разрядов или 2 байта. На 32-х разрядной машине тип **int** всегда 32 разряда или 4 байта. Таким образом, тип **int** эквивалентен типам **short int** или **long int** в зависимости от реализации.

Аналогично, тип **unsigned int** эквивалентен типам **unsigned short** или **unsigned long**. Спецификаторы типов **int** и **unsigned int** широко используются в программах на Си, поскольку они позволяют наиболее эффективно манипулировать целыми величинами на данной машине.

Однако, размер типов **int** и **unsigned int** переменный, поэтому программы, зависящие от специфики размера **int** и **unsigned int** могут быть непереносимы. Переносимость кода можно улучшить путем включения выражений с **sizeof** операцией.

Область значений величин

Область значений величин — это интервал целых величин от минимального до максимального, который может быть представлен в заданном числе бит. Например, константное выражение **-32768** состоит из арифметического отрицания **-**, предшествующего величине константы **32768**. Так как **32768** — это слишком большое значение для типа **short**, то оно задается типом **long** и следовательно константное выражение **-32768** будет иметь тип **long**. Величина **-32768** может быть представлена как **short** только путем преобразования ее типа к типу **short**. Информация не будет потеряна при преобразовании типа, поскольку **-32768** может быть представлено двумя байтами памяти. Аналогично такая величина, как **65000** может быть представлена как **unsigned short** только путем преобразования ее к типу **unsigned short** или заданием величины в восьмеричном или шестнадцатеричном представлении. Величина **65000** в

десятичном представлении рассматривается как знаковая и задается типом **long**, так как **65000** не соответствует типу **short**. Эта величина типа **long** может быть преобразована к типу **unsigned short** без потери информации, так как **65000** можно разместить в двух байтах памяти.

Восьмеричные и шестнадцатеричные константы могут быть знакового и беззнакового типа в зависимости от их размера. Однако, метод, используемый для назначения типов этим константам гарантирует, что они всегда будут преобразованы к беззнаковому целому. Числа с плавающей точкой используют IEEE (институт инженеров электриков и электронщиков) формат. Величины типа **float** занимают 4 байта, состоящих из бита знака и 7-ми битовой избыточной экспоненты и 24-х битовой мантиссы. Мантисса представляет число между 1.0 и 2.0. Так как старший бит мантиссы всегда 1, он не запоминается в памяти. Это представление дает область значений приблизительно от 3.4E-38 до 3.4E38.

Величины типа **double** занимают 8 байт. Их формат аналогичен **float** формату, за исключением того, что порядок занимает 11 бит, а мантисса 52 бита плюс бит, который опущен. Это дает область значений приблизительно от 1.7E-308 до 1.7E+308.

Деклараторы

Синтаксис:

```
<identifier>
<declarator>[]
<declarator>[constant-expression>]
*<declarator>
<declarator>()
<declarator>(<arg-type-list>)
(<declarator>)
```

Си позволяет объявлять: массивы величин, указатели на величины, величины возвратов функций. Чтобы объявить эти объекты, нужно использовать декларатор, возможно модифицированный квадратными скобками **[]**, круглыми скобками **()** и звездочкой *, что соответствует типам массива, функции или указателя. Деклараторы появляются в объявлениях указателей, массивов и функций.

Деклараторы массивов, функций и указателей

Когда декларатор состоит из немодифицируемого идентификатора, то объект, который объявляется, имеет немодифицированный тип. Звездочка, которая может появиться слева от идентификатора, модифицирует его в тип указателя. Если за идентификатором следуют квадратные скобки **[]**, то тип модифицируется на тип массива. Если за идентификатором следуют круглые скобки, то тип модифицируется на тип функции. Сам по себе декларатор не образует полного объявления. Для этого в объявление должен быть включен спецификатор типа. Спецификатор типа задает тип элементов массива или тип адресуемых объектов и возвратов функции.

Следующие примеры иллюстрируют простейшие формы деклараторов:

```
1. int list[20]
2. char *cp
3. double func(void),
```

где:

1. Массив **list** целых величин

2. Указатель **cp** на величину типа **char**

3. Функция **func** без аргументов, возвращающая величину **double**.

Составные деклараторы

Любой декларатор может быть заключен в круглые скобки. Обычно, круглые скобки используются для спецификации особенностей интерпретации составного декларатора. Составной декларатор — это идентификатор, определяемый более чем одним модификатором массива, указателя или функции.

С отдельным идентификатором могут появиться различные комбинации модификаторов массива, указателя или функции. Некоторые комбинации недопустимы. Например, массив не может быть композицией функций, а функция не может возвратить массив или функцию. При интерпретации составных деклараторов квадратные и круглые скобки (справа от идентификатора) имеют приоритет перед звездочкой (слева от идентификатора). Квадратные или круглые скобки имеют один и тот же приоритет и рассматриваются слева направо. Спецификатор типа рассматривается на последнем шаге, когда

декларатор уже полностью проинтерпретирован. Можно использовать круглые скобки, чтобы изменить порядок интерпретации на необходимый в данном случае.

При интерпретации составных деклараторов может быть предложено простое правило, которое читается следующим образом: «изнутри-наружу». Нужно начать с идентификатора и посмотреть вправо, есть ли квадратные или круглые скобки. Если они есть, то проинтерпретировать эту часть декларатора, затем посмотреть налево, если ли звездочка. Если на любой стадии справа встретится закрывающая круглая скобка, то вначале необходимо применить все эти правила внутри круглых скобок, а затем продолжить интерпретацию. На последнем шаге интерпретируется спецификатор типа. В следующем примере проиллюстрированы эти правила. Последовательность шагов при интерпретации перенумерована.

Деклараторы со специальными ключевыми словами

Можно использовать следующие специальные ключевые слова:

```
cdecl
far
fortran
huge
near
pascal
```

Эти ключевые слова используются для изменения смысла объявлений переменной и функции.

Когда специальное ключевое слово встречается в деклараторе, то оно модифицирует объект, расположенный справа от ключевого слова. Допускается более одного ключевого слова, модифицирующего объект. Например, идентификатор функции может быть модифицирован двумя ключевыми словами **far** и **pascal**. Порядок ключевых слов в этом случае не важен, то есть **far pascal** и **pascal far** имеют один и тот же смысл. В различных частях объявления могут быть использованы два или более ключевых слов для модификации смысла составляющих частей объявления.

Например, в следующем объявлении содержится в различных местах два ключевых слова **far** и ключевое слово **pascal**:

```
int far * pascal far func(void);
```

Идентификатор функции **func** модифицируется ключевыми словами **pascal** и far. func возвращает величину, объявленную как **far**-указатель на величину типа **int**.

Как и прежде в любом объявлении могут быть использованы круглые скобки для изменения порядка интерпретации. Правила интерпретации составных объявлений со специальными ключевыми словами остаются теми же, что и без них. В следующих примерах показано использование специальных ключевых слов в объявлениях.

```
\*************** Example 1 ********************\
int huge database[65000];
\*************** Example 2 *********************\
char * far * x;
\*************** Example 3 *********************\
double near cdecl calc(double,double);
double cdecl near calc(double,double);
\*************** Example 4 *********************\
char far fortran initlist[INITSIZE];
char far *nextchar, far *prevchar, far *currentchar;
\*************** Example 5 **********************\
char far *(far *getint) (int far *);
^ ^       ^      ^      ^     ^

6 5       2      1      3     4
```

В первом примере объявляется массив **huge**, поименованный **database**, содержащий 65000 элементов типа **int**. Декларатор массива модифицируется ключевым словом **huge**.

Во втором примере ключевое слово **far** модифицирует расположенную справа звездочку, делая **x far**-указателем на указатель к величине типа **char**.

Это объявление эквивалентно и такой записи:

```
char * (far *x);
```

В третьем примере показано два эквивалентных объявления. Оба объявляют calc как функции с **near** и **cdecl** атрибутами.

В четвертом примере также представлено два объявления: первое объявляет **far fortan**-массив символов, поименованный

initlist, а второе объявляет три **far**-указателя, поименованные **nexchar**, **prevchar** и **currentchar**. Указатели могут быть использованы для запоминания адресов символов массива **initlist**. Заметим, что ключевое слово **far** должно быть повторено перед каждым декларатором.

В пятом примере показано более сложное объявление с различными случаями расположения ключевого слова **far**. Интерпретация этого объявления состоит из следующих шагов:

1. Идентификатор **getint** объявляется как

2. **far**-указатель на

3. функцию, требующую

4. один аргумент, который является **far**-указателем на величину типа **int**

5. и возвращающую **far-** указатель на

6. величину типа **char**

Заметим, что ключевое слово **far** всегда модифицирует объект, который следует справа.

Объявления переменной

Объявление простой переменной

Синтаксис:

```
<type-specifier><identifier>[,<identifier>...];
```

Объявление простой переменной определяет имя переменной и ее тип; оно может также определять класс памяти переменной. Имя переменной — это идентификатор, заданный в объявлении. Спецификатор типа **type-specifier** задает имя определяемого типа данных.

Можно определить имена различных переменных в том же самом объявлении, задавая список идентификаторов, разделенных запятой. Каждый идентификатор списка именует переменную. Все переменные, заданные в объявлении, имеют один и тот же тип.

Примеры:

```
int x;    /* Example 1 */
unsigned long reply, flag     /* Example 2 */
```

```
double order;    /* Example 3 */
```

В первом примере объявляется простая переменная **x**. Эта переменная может принимать любое значение из множества значений, определяемых для типа **int**.

Во втором примере объявлены две переменные: **reply** и **flag**. Обе переменные имеют тип **unsigned long**.

В третьем примере объявлена переменная **order**, которая имеет тип **double**. Этой переменной могут быть присвоены величины с плавающей запятой.

Объявление перечисления

Синтаксис:

```
enum[<tag>]<enum-list><identifier>[,<identifier>...];
enum<tag><identifier>[,<identifier>...];
```

Объявление перечисления задает имя переменной перечисления и определяет список именованных констант, называемый списком перечисления. Значением каждого имени списка является целое число. Переменная перечисления принимает значение одной из именованных констант списка. Именованные константы списка имеют тип **int**. Таким образом, память соответствующая переменной перечисления — это память, необходимая для размещения отдельной целой величины.

Объявление перечисления начинается с ключевого слова enum и имеет две формы представления. В первой форме представления имена перечисления задаются в списке перечисления **enum-list**.

Опция **tag** — это идентификатор, который именует тип перечисления, определенного в **enum-list**.

Переменную перечисления именует **identifier**. В объявлении может быть описана более чем одна переменная перечисления.

Во второй форме используется тег перечисления, который ссылается на тип перечисления. В этой форме объявления список перечисления не представлен, поскольку тип перечисления определен в другом месте. Если задаваемый тег не ссылается на уже определенный тип перечисления, или если именуемый тегом тип находится вне текущей видимости, то выдается ошибка.

<enum-list> имеет следующий синтаксис:

```
<identifier>[=<constant-expression>][,<identifier>
[=<constant-expression]]...
.
.
.
```

Каждый идентификатор именует элементы перечисления. По умолчанию первому идентификатору соответствует значение 0, следующий идентификатор ассоциируется со значением 1 и т.д. Имя константы перечисления эквивалентно ее значению.

Запись **=<constant-expression>** переопределяет последовательность значений, заданных по умолчанию. Идентификатор, следующий перед записью **=<constant-expression>** принимает значение, задаваемое этим константным выражением. Константное выражение имеет тип **int** и может быть отрицательным. Следующий идентификатор в списке ассоциируется с величиной, равной **<constant-expression>+1**, если он явно не задается другой величиной.

Перечисление может содержать повторяющиеся значения идентификаторов, но каждый идентификатор должен быть уникальным. Кроме того, он должен быть отличным от всех других идентификаторов перечислений с той же видимостью. Например, двум различным идентификаторам **null** и **zero** может быть задано значение 0 в одном и том же перечислении. Идентификаторы должны быть отличны от других идентификаторов с той же самой видимостью, включая имена обычных переменных и идентификаторы других перечислений. Теги перечислений должны быть отличны от тегов перечислений, тегов структур и совмещений с той же самой видимостью.

Примеры:

```
/**************** Example 1 ***************/
enum day
saturday,
sunday = 0,
monday,
tuesday,
wednesday,
thursday,
friday
```

```
 workday;
/***************** Example 2 ***************/
enum day today = wednesday;
```

В первом примере определяется тип перечисления, поименованный **day** и объявляется переменная **workday** этого типа перечисления. С **saturday** по умолчанию ассоциируется значение 0. Идентификатор **sunday** явно устанавливается в 0. Оставшиеся идентификаторы по умолчанию принимают значение от 1 до 5.

Во втором примере переменной **today** типа **enum day** присваивается значение из перечисления. Заметим, что для присваивания используется имя константы из перечисления. Так как тип перечисления **day** был предварительно объявлен, то достаточно сослаться только на тег перечисления.

Объявления структур

Синтаксис:

```
struct[<tag>]<member-declaration-list><declarator>[,
<declarator>...];
struct<tag><declarator>[,<declarator>...];
```

Объявление структуры задает имя типа структуры и специфицирует последовательность переменных величин, называемых элементами структуры, которые могут иметь различные типы.

Объявление структуры начинается с ключевого слова **struct** и имеет две формы представления, как показано выше. В первой форме представления типы и имена элементов структуры специфицируются в списке объявлений элементов **<member-declaration-list>**. **<tag>** — это идентификатор, который именует тип структуры, определенный в списке объявлений элементов.

Каждый **<declarator>** задает имя переменной типа структуры. Тип переменной в деклараторе может быть модифицирован на указатель к структуре, на массив структур или на функцию, возвращающую структуру.

Вторая синтаксическая форма использует тег — **<tag>** структуры для ссылки на тип структуры. В этой форме объявления отсутствует список объявлений элементов, поскольку тип структуры определен в другом месте. Определение типа структуры должно быть видимым для тега, который используется

в объявлении и определение должно предшествовать объявлению через тег, если тег не используется для объявления указателя или структурного типа **typedef**. В последних случаях объявления могут использовать тег структуры без предварительного определения типа структуры, но все же определение должно находиться в пределах видимости объявления.

Список объявлений элементов **<member-declaration-list>** — это одно или более объявлений переменных или битовых полей. Каждая переменная, объявленная в этом списке, называется элементом структурного типа. Объявления переменных списка имеют тот же самый синтаксис, что и объявления переменных, за исключением того, что объявления не могут содержать спецификаторов класса памяти или инициализаторов. Элементы структуры могут быть любого типа: основного, массивом, указателем, совмещением или структурой.

Элемент не может иметь тип структуры, в которой он появляется. Однако, элемент может быть объявлен, как указатель на тип структуры, в которую он входит, позволяя создавать списочные структуры.

Битовые поля

Объявления битовых полей имеют следующий синтаксис:

```
<type-specifier>[<identifier>]:<constant-expression>
```

Битовое поле состоит из некоторого числа бит, специфицированных константным выражением **<constant-expression>**. Для битового поля спецификатор типа **<type-specifier>** должен специфицировать беззнаковый целый тип, а константное выражение должно быть неотрицательной целой величиной. Массивы битовых полей, указатели на битовые поля и функции, возвращающие битовые поля не допускаются. Идентификатор **<identifier>** именует битовое поле. Неименованное битовое поле, чей размер специфицируется как нулевой, имеет специальное назначение: оно гарантирует, что память для следующей переменной объявления будет начинаться на границе **int**.

Идентификаторы элементов внутри объявляемой структуры должны быть уникальными. Идентификаторы элементов внутри разных структур могут совпадать. В пределах той же самой

видимости теги структур должны отличаться от других тегов (тегов других структур, совмещений и перечислений).

Переменные (элементы) структуры запоминаются последовательно в том же самом порядке, в котором они объявляются: первой переменной соответствует самый младший адрес памяти, а последней — самый старший. Память каждой переменной начинается на границе свойственной ее типу. Поэтому могут появляться неименованные участки между соседними элементами.

Битовые поля не располагаются на пересечении границ, объявленных для них типов. Например, битовое поле, объявленное с типом **unsigned int**, упаковывается или в пространстве, оставшимся от предыдущего **unsigned int** или начиная с нового **unsigned int**.

Примеры:

```
/**************** Example 1 ****************/
struct
float x,y;
 complex;
/**************** Example 2 *****************/
struct employee
char name[20];
int id;
long class;
 temp;
/**************** Example 3 ******************/
struct employee student, faculty, staff;
/**************** Example 4 ******************/
struct sample
char c;
float *pf;
struct sample *next;
 x;
/***************** Example 5 *****************/
struct
unsigned icon : 8;
unsigned color : 4;
unsigned underline : 1;
unsigned blink : 1;
 screen[25][80];
```

В первом примере объявляется переменная с именем **complex** типа структура.

Эта структура состоит из двух элементов **x** и **y** типа **float**. Тип структуры не поименован.

Во втором примере объявляется переменная с именем **temp** типа структура. Структура состоит из трех элементов с именами **name**, **id** и **class**. Элемент с именем **name** — это массив из 20-ти элементов типа **char**. Элементы с именами **id** и **class** — это простые переменные типа **int** и **long** соответственно. Идентификатор **employee** является тегом структуры.

В третьем примере объявлены три переменных типа структура с именами: **student**, **faculty** и **staff**. Каждая из структур состоит из трех элементов одной и той же конструкции. Элементы определены при объявлении типа структуры с тегом employee в предыдущем примере.

В четвертом примере объявляется переменная с именем **x** типа структура. Первые два элемента структуры представлены переменной с типа **char** и указателем **pf** на величину типа **float**. Третий элемент с именем **next** объявляются как указатель на описываемую структуру **sample**.

В пятом примере объявляется двумерный массив поименованный **screen**, элементы которого имеют структурный тип. Массив состоит из 2000 элементов и каждый элемент — это отдельная структура, состоящая из четырех элементов типа **bit-fild** с именами **icon**, **color**, **underline** и **blink**.

Объявление совмещений

Синтаксис:

```
union[<tag>]<member-declaration-list><declarator>
[,<declarator>...];
union<tag><declarator>[,<declarator>...];
```

Объявление совмещения определяет имя переменной совмещения и специфицирует множество переменных, называемых элементами совмещения, которые могут быть различных типов. Переменная с типом совмещения запоминает любую отдельную величину, определяемую набором элементов совмещения.

Объявление совмещения имеет тот же самый синтаксис, как и объявление структуры, за исключением того, что она начинается с ключевого слова **union** вместо ключевого слова **struct**. Для объявления совмещения и структуры действуют одни и те же правила, за исключением того, что в совмещении не допускаются элементы типа битовых полей.

Память, которая соответствует переменной типа совмещение, определяется величиной для размещения любого отдельного элемента совмещения.

Когда используется наименьший элемент совмещения, то переменная типа совмещения может содержать неиспользованное пространство. Все элементы совмещения запоминаются в одном и том же пространстве памяти переменной, начиная с одного и того же адреса. Запомненные значения затираются каждый раз, когда присваивается значение очередного элемента совмещения.

Объявление массива

Синтаксис:

```
<type-specifier><declarator>[<constant-expression>];
<type-specifier><declarator>[];
```

Здесь квадратные скобки — это терминальные символы. Объявление массива определяет тип массива и тип каждого элемента. Оно может определять также число элементов в массиве. Переменная типа массив рассматривается как указатель на элементы массива. Объявление массива может представляться в двух синтаксических формах, указанных выше.

Декларатор **<declarator>** задает имя переменной. Квадратные скобки, следующие за декларатором, модифицируют декларатор на тип массива. Константное выражение **<constant-expression>**, заключенное в квадратные скобки, определяет число элементов в массиве. Каждый элемент имеет тип, задаваемый спецификатором типа **<type-specifier>**, который может специфицировать любой тип, исключая **void** и тип функции.

Во второй синтаксической форме опущено константное выражение в квадратных скобках. Эта форма может быть использована только тогда, когда массив инициализируется или объявлен как формальный параметр или объявлен как ссылка на массив, явно определенный где-то в программе.

Массив массивов или многомерный массив определяется путем задания списка константных выражений в квадратных скобках, следующего за декларатором:

```
<type-specifier><declarator>[<constant-expression>]
[<constant-expression>]...
```

Каждое константное выражение **<constant-expression>** в квадратных скобках определяет число элементов в данном измерении массива, так что объявление двумерного массива содержит два константных выражения, трехмерного — три и т.д. Если многомерный массив объявляется внутри функции или если он инициализируется либо объявляется как формальный параметр или объявляется как ссылка на массив, явно определенный где-то в программе, то первое константное выражение может быть опущено.

Массив указателей на величины заданного типа может быть определен посредством составного декларатора.

Типу «массив» соответствует память, которая требуется для размещения всех его элементов. Элементы массива с первого до последнего запоминаются в последовательных возрастающих адресах памяти. Между элементами массива в памяти разрывы отсутствуют. Элементы массива запоминаются друг за другом построчно. Например, массив, содержащий две строки с тремя столбцами каждая, **char A[2][3]** будет запомнен следующим образом. Сначала запоминаются три столбца первой строки, затем элементы трех столбцов второй строки. Смысл этого в том, чтобы последний индекс был более быстрым. Чтобы сослаться на отдельный элемент массива, нужно использовать индексное выражение.

Объявление указателей

Синтаксис:

```
<type-specifier> *<declarator>
```

Объявление указателя определяет имя переменной типа указатель и тип объекта, на который указывает эта переменная. Декларатор **<declarator>** определяет имя переменной с возможной модификацией ее типа. Спецификатор типа **<type- specifier>** задает тип объекта, который может быть базового типа, типа структуры или совмещения.

Переменная типа указатель может указывать также на функции, массивы и другие указатели.

Если указатель не используется до определения типа структуры или совмещения, то он может быть объявлен ранее этого определения. Такие объявления допускаются, поскольку компилятору не требуется знать размер структуры или совмещения, чтобы распределить память под переменную типа указатель. Указатель может быть объявлен посредством использования тега структуры или совмещения.

Переменная, объявленная как указатель, хранит адрес памяти. Размер памяти, требуемый для адреса, и смысл адреса зависит от данной конфигурации машины. Указатели на различные типы не обязательно имеют одну и ту же длину.

Для некоторых реализаций используются специальные ключевые слова **near**, **far** и **huge**, чтобы модифицировать размер указателя.

Как объявляются функции

Синтаксис:

```
[<type-specifier>]<declarator>([<arg-type-list>])
[,<declarator>...];
```

Объявление функции определяет имя, тип возврата функции и, возможно, типы и число ее аргументов. Объявление функции также называется **forward**-объявлением. Декларатор функции объявляет имя функции, а спецификатор типа задает тип возврата. Если спецификатор типа опущен в объявлении функции, то предполагается, что функция возвращает величину типа **int**.

Объявление функции может включать спецификаторы класса памяти **extern** или **static**.

Список типов аргументов

Список типов аргументов **<arg-type-list>** определяет число и типы аргументов функции. Синтаксис списка аргументов следующий:

```
<type-name-list>[,...]
```

Список имен типов — это список из одного или более имен типов. Каждое имя типа отделяется от другого запятой. Первое имя типа задает тип первого аргумента, второе имя типа задает

тип второго аргумента и т. д. Если список имен типов заканчивается запятой с многоточием (**,...**), то это означает, что число аргументов функции переменно. Однако, предполагается, что функция будет иметь не меньше аргументов, чем имен типов, предшествующих многоточию.

Если список типов аргументов **<arg-type-list>** содержит только многоточие, то число аргументов функции является переменным или равно нулю.

Важно: Чтобы поддержать совместимость с программами предыдущих версий, компилятор допускает символ запятой без многоточия в конце списка типов аргументов для обозначения их переменного числа. Запятая может быть использована и вместо многоточия для объявления нуля или более аргументов функции. Использование запятой поддерживается только для совместимости. Использование многоточия рекомендуется для нового представления.

Имя типа **<type-name>** для типов структуры, совмещения или базового типа состоит из спецификатора этого типа (такого как **int**). Имена типов для указателей, массивов и функций формируются путем комбинации спецификатора типа с абстрактным декларатором. Абстрактный декларатор — это декларатор без идентификатора.

Для того чтобы объявить функцию, не имеющую аргументов, может быть использовано специальное ключевое слово **void** на месте списка типов аргументов. Компилятор вырабатывает предупреждающее сообщение, если в вызове такой функции будут специфицированы аргументы.

Еще одна специальная конструкция допускается в списке типов аргументов. Это фраза **void ***, которая специфицирует аргумент типа указатель. Эта фраза может быть использована в списке типов аргументов вместо имени типа.

Список типов аргументов может быть опущен. В этом случае скобки после идентификатора функции все же требуются, хотя они и пусты. В этом случае в объявлении функции не определяются ни типы, ни число аргументов в функции. Когда эта информация опускается, то компилятор не проверяет соответствия между формальными и фактическими параметрами при вызове функции.

Тип возврата

Функции могут возвращать величины любого типа за исключением массивов и функций. Для этого посредством спецификатора типа **type-specifier** в объявлении функции можно специфицировать любой тип: основной, структуру или совмещение. Идентификатор функции может быть модифицирован одной или несколькими звездочками (*), чтобы объявить возвращаемую величину типа указателя.

Хотя функции и не допускают возвратов массивов или функций, но они могут возвращать указатели на массивы или функции. Функции, которые возвращают указатели на величины типа массив или функция, объявляются посредством модификации идентификатора функции квадратными скобками, звездочкой и круглыми скобками, чтобы сформировать составной декларатор.

Классы памяти

Класс памяти переменной, которая определяет какой либо объект, имеет глобальное или локальное время жизни. Объект с глобальным временем жизни существует и имеет значение на протяжении всей программы. Все функции имеют глобальное время жизни.

Переменные с локальным временем жизни захватывают новую память при каждом выполнении блока, в котором они определены. Когда управление на выполнение передается из блока, то переменная теряет свое значение.

Хотя Си определяет два типа классов памяти, но, тем не менее, имеется следующих четыре спецификатора классов памяти:

```
auto
register
static
extern
```

Объекты классов **auto** и **register** имеют локальное время жизни. Спецификаторы **static** и **extern** определяют объекты с глобальным временем жизни. Каждый из спецификаторов класса памяти имеет определенный смысл, который влияет на видимость функций и переменных в той же мере, как и сами классы памяти.

Термин «видимость» относится к той части программы, в которой могут ссылаться друг на друга функции и переменные. Объекты с глобальным временем жизни существуют на протяжении выполнения исходной программы, но они могут быть видимы не во всех частях программы.

Месторасположение объявления переменной или функции внутри исходных файлов также влияют на класс памяти и видимость. Говорят, что объявления вне определения всех функций и переменных относятся к внешнему уровню, а объявления внутри определений функций относятся к внутреннему уровню.

Точный смысл каждого спецификатора класса памяти зависит от того, находится ли объявление на внешнем или внутреннем уровне и от того, объявлен ли объект функцией или переменной.

Объявления переменной на внешнем уровне

Объявления переменной на внешнем уровне используют спецификаторы класса памяти **static** и **extern** или совсем опускают их. Спецификаторы класса памяти auto и **register** не допускаются на внешнем уровне.

Объявления переменных на внешнем уровне — это определения переменных или ссылки на определения, сделанные в другом месте.

Объявление внешней переменной, которое инициализирует эту переменную (явно или неявно), называется определением этой переменной. Определение на внешнем уровне может задаваться в различных формах.

Переменная на внешнем уровне может быть определена путем ее объявления со спецификатором класса памяти **static**. Такая переменная может быть явно инициализирована константным выражением. Если инициализатор отсутствует, то переменная автоматически инициализируется нулем во время компиляции. Таким образом, объявления **static int k = 16** и **static int k** — оба рассматриваются как определения.

Переменная определяется, когда она явно инициализируется на внешнем уровне. Например, **int j = 3** — это определение переменной.

Так как переменная определяется на внешнем уровне, то она видима в пределах остатка исходного файла, от места, где она определена. Переменная не видима выше своего определения в том же самом исходном файле ни в других исходных файлах программы, если не объявлена ссылка, которая делает ее видимой.

Переменная может быть определена на внешнем уровне внутри исходного файла только один раз. Если задается спецификатор класса памяти **static**, то в других исходных файлах могут быть определены переменные с тем же именем. Так как каждое определение **static** видимо только в пределах своего собственного исходного файла, то конфликта не возникнет.

Спецификатор класса памяти **extern** используется для объявления ссылки на переменную, определенную где-то в другом месте. Такие объявления используются в случае, когда нужно сделать видимым определение переменной в других исходных файлах или выше места, где она определена в том же самом исходном файле. Так как ссылка на переменную объявлена на внешнем уровне, то переменная видима в пределах остатка исходного файла от места объявления ссылки.

В объявлениях, которые используют спецификатор класса памяти **extern**, инициализация не допускается, так как они ссылаются на переменные, чьи величины уже определены.

Переменная, на которую делается ссылка **extern**, должна быть определена на внешнем уровне только один раз. Определение может быть сделано в любом из исходных файлов, составляющих программу.

Есть одно исключение из правил, описанных выше. Можно опустить из объявления переменной на внешнем уровне спецификатор класса памяти и инициализатор. Например, объявление **int n;** будет правильным внешним объявлением. Это объявление имеет два различных смысла в зависимости от контекста.

1. Если где-нибудь в программе будет определена на внешнем уровне переменная с тем же именем, то объявление является ссылкой на эту переменную, как если бы был использован спецификатор класса памяти **extern** в объявлении.

2. Если нет такого определения, то объявленной переменной распределяется память во время линкования и переменная инициализируется нулем. Если в программе появится более чем одно такое объявление, то память распределится для наибольшего размера из объявленных переменных. Например, если программа содержит два неинициализированных объявления переменной **i** на внешнем уровне **int i;** и **char i;** то память во время линкования распределится под переменную **i** типа **int**.

Неинициализированные объявления переменной на внешнем уровне не рекомендуются для файлов, которые могут быть размещены в библиотеку.

Пример:

```
/**********************************************
SOURCE FILE ONE
**********************************************/
extern int i;   /* reference to i
defined below */
main()

i++;
printf("%d\n", i);     /* i equals 4 */
next();

int i = 3;      /* definition of i */
next()
i++;
printf("%d\n", i);     /* i equals 5 */
other();

/**********************************************
SOURCE FILE TWO
**********************************************/
extern int i;   /* reference to i in
first source file */
other()

i++;
printf("%d\n", i);     /* i equals 6 */
```

Объявление переменной на внутреннем уровне

Любой из четырех спецификаторов класса памяти может быть использован для объявления переменной на внутреннем уровне. Если спецификатор класса памяти опускается в объявлении переменной на внутреннем уровне, то подразумевается класс памяти **auto**.

Спецификатор класса памяти **auto** объявляет переменную с локальным временем жизни. Переменная видима только в том блоке, где она объявлена. Объявления переменных **auto** могут включать инициализаторы. Переменные класса памяти **auto** автоматически не инициализируются, а инициализируются явно при объявлении или присваивании начальных значений, посредством операторов внутри блока. Если нет инициализации, то величина переменной **auto** считается неопределенной.

Спецификатор класса памяти **register** сообщает компилятору о том, чтобы он распределил память под переменную в регистре, если это возможно. Использование регистровой памяти обычно приводит к более быстрому времени доступа и к меньшему размеру результирующего кода. Переменные, объявленные с классом памяти **register** имеют ту же самую видимость, что и переменные **auto**.

Число регистров, которое может быть использовано под память переменных, зависит от машины. Когда компилятор встречает спецификатор класса памяти **register** в объявлении, а свободного регистра не имеется, то для переменной распределяется память класса **auto**. Компилятор назначает переменным регистровую память в том порядке, в котором появляются объявления в исходном файле. Регистровая память, если она имеется, гарантирована только для целого и адресного типов.

Переменная, объявленная на внутреннем уровне со спецификатором класса памяти **static**, имеет глобальное время жизни и имеет видимость только внутри блока, в котором она объявлена. В отличие от переменных **auto**, переменные, объявленные как **static**, сохраняют свое значение при завершении блока.

Переменные класса памяти **static** могут быть инициализированы константным выражением. Если явной инициализации нет, то переменная класса памяти **static**

автоматически устанавливается в **0**. Инициализация выполняется один раз во время компиляции. Инициализация переменной класса памяти **static** не повторяется при новом входе в блок.

Переменная, объявленная со спецификатором класса памяти **extern**, является ссылкой на переменную с тем же самым именем, определенную на внешнем уровне в любом исходном файле программы.

Цель внутреннего объявления **extern** состоит в том, чтобы сделать определение переменной внешнего уровня видимой внутри блока. Внутреннее объявление **extern** не изменяет видимость глобальной переменной в любой другой части программы.

Пример:

```
int i = 1;
main()
   /* reference to i, defined above */
extern int i;
/* initial value is zero; a is
visible only within main */
static int a;
/* b is stored in a register, if possible */
register int b = 0;
/* default storage class is auto */
int c = 0;
/* values printed are 1, 0, 0, 0 */
printf("%d\n%d\n%d\n%d\n", i, a, b, c);
other();

other()

/* i is redefined */
int i = 16;
/* this a is visible only within other */
static int a = 2;
a += 2;
/* values printed are 16, 4 */
printf("%d\n%d\n", i, a);
```

Переменная **i** определяется на внешнем уровне с инициализацией **1**. В функции **main** объявлена ссылка **extern** на

переменную **i** внешнего уровня. Переменная класса памяти **static** автоматически устанавливается в **0**, так как инициализатор опущен. Вызов функции **print** (предполагается, что функция **print** определена в каком-то месте исходной программы) печатает величины **1**, **0**, **0**, **0**.

В функции **other**, переменная **i** переопределяется как локальная переменная с начальным значением **16**. Это не влияет на значение внешней переменной **i**. Переменная **a** объявляется как переменная класса памяти **static** с начальным значением **2**. Она не противоречит переменной **a**, объявленной в функции **main**, так как видимость переменных класса памяти **static** на внутреннем уровне ограничена блоком, в котором она объявлена.

Объявление функции на внешнем и внутреннем уровнях

Функции могут быть объявлены со спецификаторами класса памяти **static** или **extern**. Функции всегда имеют глобальное время жизни.

Правила видимости для функций отличаются от правил видимости для переменных. Объявления функций на внутреннем уровне имеют тот же самый смысл, что и объявления на внешнем уровне. Это значит, что функции не могут иметь блочной видимости и видимость функций не может быть вложенной. Функция объявленная как **static**, видима только в пределах исходного файла, в котором она определяется. Любая функция в том же самом исходном файле может вызвать функцию **static**, но функции **static** из других файлов нет. Функция **static** с тем же самым именем может быть объявлена в другом исходном файле.

Функции, объявленные как **extern** видимы в пределах всех исходных файлов, которые составляют программу. Любая функция может вызвать функцию **extern**.

Объявления функций, в которых опущен спецификатор класса памяти, считаются по умолчанию **extern**.

Инициализация

В объявлении переменной может быть присвоено начальное значение посредством инициализатора. Величина или величины инициализатора присваиваются переменной.

Синтаксически записи инициализатора предшествует знак

равно (=):

```
=<initializer>
```

Могут быть инициализированы переменные любого типа. Функции не инициализируются. Объявления, которые используют спецификатор класса памяти **extern** не могут содержать инициализатора.

Переменные, объявленные на внешнем уровне, могут быть инициализированы. Если они явно не инициализированы, то они устанавливаются в нуль во время компиляции или линкования. Любая переменная, объявленная со спецификатором класса памяти **static**, может быть инициализирована константным выражением. Инициализация переменных класса **static** выполняется один раз во время компиляции. Если отсутствует явная инициализация, то переменные класса памяти **static** автоматически устанавливаются в нуль.

Инициализация переменных **auto** и **register** выполняется каждый раз при входе в блок, в котором они объявлены. Если инициализатор опущен в объявлении переменной класса памяти **auto** или **register**, то начальное значение переменной не определено. Инициализация составных типов **auto** (массив, структура, совмещение) запрещена. Любое составное объявление класса памяти **static** может быть инициализировано на внешнем уровне.

Начальными значениями для внешних объявлений переменной и для всех переменных **static** как внешних так и внутренних должно быть константное выражение. Автоматические и регистровые переменные могут быть инициализированы константными или переменными величинами.

Базовые типы и типы указателей

Синтаксис:

```
=<expression>
```

Величина выражения присваивается переменной. Для выражения допустимы правила преобразования.

Составные типы

Синтаксис:

`=<initializer-list>`

Список инициализаторов **<initializer-list>** — это последовательность инициализаторов, разделенных запятыми. Каждый инициализатор в последовательности — это либо константное выражение, либо список инициализаторов. Поэтому, заключенный в фигурные скобки список, может появиться внутри другого списка инициализации. Эта конструкция используется для инициализации элементов составных конструкций.

Для каждого списка инициализации значения константных выражений присваиваются в порядке следования элементов составной переменной. Когда инициализируется совмещение, то список инициализаторов представляет собой единственное константное выражение. Величина константного выражения присваивается первому элементу совмещения.

Если в списке инициализации меньше величин, чем их имеется в составном типе, то оставшиеся памяти инициализируются нулем. Если число инициализирующих величин больше чем требуется, то выдается ошибка.

Эти правила применяются к каждому вложенному списку инициализаторов, точно так же как и ко всей конструкции в целом.

Эти дополнительные запятые допускаются, но не требуются. Требуются только те запятые, которые разделяют константные выражения и списки инициализации. Если список инициализаторов не структурирован под составной объект, то его величины присваиваются в том порядке, в котором подстыкованы элементы объекта.

Фигурные скобки могут также появляться вокруг индивидуальных инициализаторов в списке.

Когда инициализируются составные переменные, то нужно позаботиться о том, чтобы правильно использовать фигурные скобки и списки инициализаторов. В следующем примере иллюстрируется более детально интерпретация компилятором фигурных скобок.

Строковые инициализаторы

Массив может быть инициализирован строчным литералом.

Например,

```
char code[ ] = "abc";
```

инициализирует **code** как массив символов из четырех элементов. Четвертым элементом является символ **\0**, который завершает все строковые литералы.

Если специфицируется размер массива, а строка больше чем специфицированный размер, то лишние символы отбрасываются. Следующее объявление инициализирует переменную **code**, как трехэлементный массив символов:

```
char code[3] = "abcd"
```

В примере только три первые символа инициализатора назначаются для массива **code**. Символ **d** и символ нуль отбрасываются.

Если строка короче, чем специфицированный размер массива, то оставшиеся элементы массива инициализируются нулем (символом **\0**).

Объявления типов

Объявление типа определяет имя и элементы структурного или совмещающего типов или имя и перечислимое множество перечислимого типа.

Имя типа может быть использовано в объявлениях переменных и функций в качестве ссылки на этот тип. Это полезно, когда многие переменные или функции имеют один и тот же тип.

Объявление **typedef** определяет спецификатор типа для типа. Это объявление используется для того, чтобы создавать более короткие или более осмысленные имена типов уже определенных в Си или объявленных пользователем.

Типы структур, совмещений и перечислений

Объявления типов структур, совмещений и перечислений имеют ту же самую общую синтаксическую форму, как и объявления переменных этих типов. В объявлении типа идентификатор переменной опущен, так как нет переменной

которая объявляется. Именем структуры, совмещения или перечисления является тег.

В объявлении типа может появиться список объявлений элементов **<member-declaration-list>** или список перечисления **<enum-list>**, определяющие тип.

Сокращенная форма объявления переменной, в котором **tag** ссылается на тип, определенный где-то еще, при объявлении типа не используется.

Объявления typedef

Синтаксис:

```
typedef <type-spesifier><declarator>[,<declarator>...];
```

Объявления **typedef** являются аналогом объявления переменной, за исключением того, что ключевое слово **typedef** заменяет спецификатор класса памяти.

Объявление интерпретируется тем же самым путем, как объявления переменной или функции, но **<declarator>** вместо того, чтобы стать переменной типа, специфицированного объявлением, становится синонимом имени типа. Объявление **typedef** не создает типов. Оно создает синонимы для существующих имен типов, которые были специфицированы другим способом. Любой тип может быть объявлен с **typedef**, включая типы указателя, функции и массива. Имя с ключевым словом **typedef** для типов указателя, структуры или совмещения может быть объявлено прежде чем эти типы будут определены, но в пределах видимости объявления.

Имена типов

Имя типа специфицирует особенности типа данных. Имена типов используются в трех контекстах: в списках типов аргументов, при объявлении функций, в вычислениях **cast** (преобразованиях типов), и в **sizeof** операциях.

Именами для основных, перечисляющих, структурных и совмещающих типов являются спецификаторы типа для каждого из них. Имена для типов указателя, массива и функции задаются следующей синтаксической формой:

```
<type-specifier><abstract-declarator>
```

Абстрактный декларатор **<abstract-declarator>** — это декларатор без идентификатора, состоящий из одного или более модификаторов указателей, массивов и функций. Модификатор указателя ***** всегда появляется перед идентификатором в деклараторе, в то время как модификатор массива **[]** или функции **()** появляются после идентификатора. Таким образом, чтобы правильно интерпретировать абстрактный декларатор, нужно начинать интерпретацию с подразумеваемого идентификатора.

Абстрактные деклараторы могут быть составными. Скобки в составном абстрактном деклараторе специфицируют порядок интерпретации, подобно тому как это делается при интерпретации составных деклараторов объявлений. Абстрактный декларатор, состоящий из пустых круглых скобок **()** не допускается, поскольку это двусмысленно. В этом случае невозможно определить находится ли подразумеваемый идентификатор внутри скобок, и в таком случае — это немодифицированный тип, или перед скобками, тогда — это тип функции. Спецификаторы типа, установленные посредством объявлений **typedef**, также рассматриваются как имена типов.

Выражения и присваивания

В Си присваиваются значения выражений. Помимо простого присваивания посредством операции **=**, Си поддерживает составные операции присваивания, которые перед присваиванием выполняют дополнительные операции над своими операндами. Окончательное значение результата зависит от старшинства операций и от побочных эффектов, если они возникают. Порядок вычисления устанавливается определенным группированием операндов и операций в выражении. Побочный эффект — это изменения состояния машины, вызванные в процессе вычисления выражения.

В выражении с побочным эффектом, вычисление одного операнда может зависеть от значения другого. Для одних и тех же операций от порядка, в котором вычисляются операнды, также зависит результат выражения.

Величина, представляемая каждым операндом в выражении, имеет тип, который может быть преобразован к другим типам в определенном контексте. Преобразования типов имеют место в

присваиваниях, **cast** операциях, вызовах функций и при выполнении операций.

Операнды

Операнд в Си — это константа, идентификатор, строка, вызов функции, индексное выражение, выражение выбора структурного элемента или более сложное выражение, сформированное комбинацией операндов и операций или заключением операндов в скобки. Любой операнд, который имеет константное значение, называется константным выражением.

Каждый операнд имеет тип. Операнд может быть преобразован из оригинального типа к другому типу посредством операции преобразования типов. Выражение преобразования типа может быть использовано в качестве операнда выражения.

Константы

Операнду-константе соответствует значение и тип представляющей его константы. Константа-символ имеет тип **int**. Целая константа имеет типы: **int**, **long**, **unsigned int** или **unsigned long**, в зависимости от размера целого и от того как специфицирована его величина. Константы с плавающей точкой всегда имеют тип **double**.

Идентификаторы

Идентификаторы именуют переменные и функции. Каждый идентификатор имеет тип, который устанавливается при его объявлении. Значение идентификатора зависит от типа следующим образом:

- идентификаторы целых и плавающих типов представляют величины соответствующего типа.
- идентификатор перечисляющего типа представляет значение одной константы из множества значений констант в перечислении. Значение идентификатора равно значению этой константы. Тип значения есть **int**, что следует из определения перечисления.
- идентификатор структурного или совмещающего типов представляет величины, специфицированные в структуре или совмещении.

- идентификатор, объявленный как указатель, представляет указатель на величину специфицированного типа.
- идентификатор, объявленный как массив, представляет указатель, чье значение является адресом первого элемента массива. Тип адресуемых указателем величин — это тип элементов массива. Например, если **series** объявлен как массив целых из 10-ти элементов, то идентификатор **series** представляет адрес массива, тогда как индексное выражение s**eries[5]** ссылается на шестой элемент массива. Адрес массива не изменяется во время выполнения программы, хотя значения отдельных элементов могут изменяться. Значение указателя, представленное идентификатором массива, не является переменной и поэтому идентификатор массива не может появляться в левой части операции присваивания.
- идентификатор, объявленный как функция, представляет указатель, чье значение является адресом функции. Тип, адресуемый указателем, — это специфицированный тип функционального возврата. Адрес функции не изменяется во время выполнения программы. Меняется только значение возврата. Таким образом, идентификаторы функции не могут появляться в левой части операции присваивания.

Строки

Строковый литерал состоит из последовательности символов, заключенных в двойные кавычки. Строковый литерал представляется в памяти как массив элементов типа **char**. Строковый литерал представляет адрес первого элемента этого массива. Адрес первого элемента строки является константой, так же как и сама строка.

Так как строковые литералы — это полноценные указатели, то они могут быть использованы в контексте, допускающем величины типа указателей, подчиняясь при этом тем же самым ограничениям. Строковые литералы имеют все же одно дополнительное ограничение: они не изменяемы и не могут появиться в левой части операции присваивания.

Последним символом строки всегда является символ нуль **0**.

Символ нуль не видим в строковом выражении, но он добавляется как последний элемент, когда строка запоминается. Таким образом, строка **abc** содержит четыре символа, а не три.

Вызовы функций

Синтаксис:

```
<expression>(<expression-lils>)
```

Вызов функции состоит из выражения **<expression>**, за которым следует список выражений **<expression-list>**. Значению выражения соответствует адрес функции (например, значение идентификатора функции). Значение каждого выражения из списка выражений (выражения в списке разделены запятыми) соответствует фактическому аргументу функции. Список выражений может быть пустым.

Выражение вызова функции имеет значение и тип своего возврата. Если тип возврата функции **void**, то и выражение вызова функции имеет тип **void**. Если возврат из вызванной функции произошел не в результате выполнения оператора **return**, то значение функции не определено.

Индексные выражения

Синтаксис:

```
<expression1>[<expression2>]
```

Здесь квадратные скобки — это терминальные символы. Индексное выражение представляет величину, адрес которой состоит из суммы значений выражения1 **<expression1>** и выражения2 — **<expression2>**. Выражение1 — это любой указатель, такой как идентификатор массива, а выражение2 — это целочисленная величина. Выражение2 должно быть заключено в квадратные скобки **[]**.

Индексное выражение обычно используется для ссылок на элементы массива, тем не менее, индекс может появиться с любым указателем.

Индексное выражение вычисляется путем сложения целой величины **<expression2>** с значением указателя **<expression1>** с последующим применением к результату операции разадресации *. Например, для одномерного массива следующие четыре

выражения эквивалентны в предположении, что **a** — это указатель, а **b** — это целое.

```
a[b]
*(a + b)
*(b + a) b[a]
```

В соответствии с правилами преобразования типов для операции сложения, целочисленная величина преобразуется к адресному представлению путем умножения ее на размер типа, адресуемого указателем. Например, предположим, что идентификатор **line** ссылается на массив величин типа **int**. Чтобы вычислить выражение **line[i]**, целая величина **i** умножается на размер типа **int**. Преобразованное значение **i** представляет **i** позиций типа **int**. Это преобразованное значение складывается с начальным значением указателя **line**, что дает адрес, который расположен на **i** позиций типа **int** от **line**.

Последним шагом вычисления индексного выражения является операция разадресации, применяемая к полученному адресу. Результатом является значение элемента массива, который позиционирован.

Заметим, что индексное выражение **line[0]** представляет значение первого элемента массива, так как отсчет смещения ведется от нуля.

Следовательно, такое выражение, как **line[5]**, ссылается на шестой элемент массива.

Ссылки на многомерный массив

Индексное выражение может быть снова проиндексировано. Синтаксис такого выражения следующий:

```
<expression1>[<expression2>][<expression3>]...
```

Данное индексное выражение интерпретируется слева направо. Сначала вычисляется самое левое индексное выражение **<expression1>[<expression2>]**. Адрес результата сложения **<expressin1>** и **<expression2>** имеет смысл адресного выражения, с которым складывается **<expression3>** и т.д. Операция разадресации осуществляется после вычисления последнего индексного выражения. Однако, операции разадресации не производится, если значение последнего указателя адресует величину типа массив.

Выражения с несколькими индексами ссылаются на элементы многомерных массивов. Многомерный массив — это массив, элементами которого являются массивы. Например, первым элементом трехмерного массива является массив с двумя измерениями.

Выражения с операциями

Выражения с операциями могут быть унарными, бинарными или тернарными. Унарное выражение состоит из операнда с предшествующей унарной операцией **<unop>** или операнда, заключенного в круглые скобки, с предшествующим ему ключевым словом **sizeof**.

Синтаксис:

```
<unop><operand>
sizeof<operand>
```

Бинарное выражение состоит из двух операндов, разделенных бинарной операцией **<binop>**.

Синтаксис:

```
<operand><binop><operand>
```

Тернарное выражение состоит из трех операндов, разделенных тернарной операцией **?:**

Синтаксис:

```
<operand> ? <operand> : <operand>
```

Выражения присваивания используют унарные, бинарные и составные операции присваивания. Унарными операциями присваивания являются инкремент **++** и декремент **--**. Бинарная операция присваивания всего одна **=**. Составные операции присваивания будем обозначать как **<compound-assign-ops>**. Каждая составная операция присваивания — это комбинация бинарной операции с простой операцией присваивания.

Синтаксис выражений присваивания:

```
<operand> ++
<operand> --
++ <operand>
-- <operand>
<operand> = <operand>
<operand> <compound-assignment-ops> <operand>
```

Выражения в скобках

Любой операнд может быть заключен в скобки. Они не влияют на тип и значение выражения, заключенного в скобки. Например, в выражении **(10 + 5) / 5** скобки, заключающие запись **10 + 5**, означают, что величина **10 + 5** является левым операндом операции деления. Результат выражения **(10 + 5) / 5** равен **3**. Без скобок значение записи **10 + 5 / 5** равнялось бы **11**. Хотя скобки влияют на то, каким путем группируются операнды в выражении, они не гарантируют детальный порядок вычисления выражения.

Type-cast выражения

Type-cast выражения имеют следующий синтаксис:

```
(<type-name>)<operand>
```

Константные выражения

Константное выражение — это выражение, результатом вычисления которого является константа. Операндами константного выражения могут быть целые константы, константы символы, плавающие константы, константы перечисления, **type-cast** выражения целого и плавающего типов и другие константные выражения. Операнды могут комбинироваться и модифицироваться посредством операций.

Константные выражения не могут использовать операции присваивания или бинарную операцию последовательного вычисления. Унарная операция адресации **&** может быть использована только при некоторых инициализациях.

Константные выражения, используемые в директивах препроцессора, имеют дополнительные ограничения, поэтому называются ограниченнымиконстантными выражениями:

```
<restricted-constant-expression>
```

Ограниченные константные выражения не могут содержать **sizeof**-выражений, констант перечисления или **type-cast** выражений любого типа. Они могут, однако, содержать специальные константные, имеющие синтаксис:

```
defined(<identifier>)
```

Эти дополнительные ограничения также относятся к константным выражениям, используемым для инициализации переменных на внешнем уровне. Тем не менее, такие выражения допускают применение унарной операции адресации **&** к любой

переменной внешнего уровня с основными и структурными типами, а также с типом совмещения и массивом внешнего уровня, индексированным константным выражением.

В этих выражениях допускается сложение или вычитание с адресными подвыражениями.

Операции

Си-операции требуют один операнд (унарные операции), два операнда (бинарные операции) или три операнда (тернарная операция). Операции присваивания — это унарные или бинарные операции.

Унарными операциями Си являются следующие:

- ~ !

Операции дополнения.

* &

Операции разадресации и адресации.

sizeof

size-операция.

Интерпретация унарных операций производится справа налево. Бинарные операции интерпретируются слева направо. Бинарными операциями являются следующие:

* / %

Мультипликативные операции.

+ -

Аддитивные операции.

<< >>

Операции сдвига.

< > <= >= == !=

Операции отношений.

& | ^

Операции с битами.

,

Операция последовательных вычислений.

&& |

Логические операции.

В Си имеется одна тернарная операция — это операция условия **?:**. Она интерпретируется справа налево.

Обычные арифметические преобразования

Большинство операций Си выполняют преобразование типов, чтобы привести операнды выражений к общему типу, или чтобы расширить короткие величины до размера целых величин, используемых в машинных операциях. Преобразования, выполняемые операциями Си, зависят от специфики операций и от типа операнда или операндов. Тем не менее, многие операции выполняют похожие преобразования целых и плавающих типов. Эти преобразования известны как арифметические преобразования, поскольку они применяются к типам величин, обычно используемых в арифметике.

Арифметические преобразования, приведенные ниже, называются обычные арифметические преобразования.

Обычные арифметические преобразования осуществляются следующим образом:

1. Операнды типа **float** преобразуются к типу **double**.

2. Если один операнд типа **double**, то второй операнд преобразуется к типу **double**.

3. Любые операнды типов **char** или **short** преобразуются к **int**.

4. Любые операнды типов **unsigned char** или **unsigned short** преобразуются к типу **unsigned int**.

5. Если один операнд типа **unsigned long**, то второй операнд преобразуется к типу **unsigned long**.

6. Если один операнд типа **long**, то второй операнд преобразуется к типу **long**.

7. Если один операнд типа **unsigned int**, то второй операнд преобразуется к **unsigned int**.

Операции дополнения

Арифметическое отрицание

Операция арифметического отрицания вырабатывает отрицание своего операнда. Операнд должен быть целой или

плавающей величиной. При выполнении операции осуществляются обычные арифметические преобразования.

Двоичное дополнение (~)

Операция двоичного дополнения вырабатывает двоичное дополнение своего операнда. Операнд должен быть целого типа. Обычные арифметические преобразования осуществляются. Результат имеет тип преобразованного операнда.

Логическое не (!)

Операция логического не (**!**) вырабатывает значение **0**, если операнд есть **true** и значение **1**, если операнд есть **false**. Результат имеет тип **int**. Операнд должен быть целого, плавающего или адресного типа.

Операция адресации и разадресации

Разадресация (*)

Разадресуемая величина доступна операции разадресации через указатель. Операнд должен быть указателем величины. Результатом операции является величина, на которую указывает операнд. Типом результата является тип, адресуемый указателем. Если значение указателя равно нулю, то результат непредсказуем.

Адресация (&)

Операция адресации (**&**) вырабатывает адрес своего операнда. Операндом может быть любая величина, которая допустима в качестве любого операнда операции присваивания.

Результат операции адресации является указателем на операнд. Тип, адресуемый указателем, является типом операнда.

Операция адресации не может применяться к битовым полям памяти структур, она не может быть применена также к идентификаторам класса памяти **register**.

Операция sizeof

Операция **sizeof** определяет размер памяти, который соответствует идентификатору или типу. Выражение **sizeof** имеет форму:

```
sizeof(<name>)
```

где **<name>** — или идентификатор или имя типа. Имя типа не может быть **void**. Значением выражения **sizeof** является размер

памяти в байтах, соответствующий поименованному идентификатору или типу.

Когда операция **sizeof** применяется к идентификатору массива, то результатом является размер всего массива в байтах, а не размер указателя, соответствующего идентификатору массива.

Когда операция **sizeof** применяется к тегу типа структуры или совмещения или к идентификатору, имеющему тип структуры или совмещения, то результатом является фактический размер в байтах структуры или совмещения, который может включать участки пространства, используемые для выравнивания элементов структуры или совмещения на границы физической памяти. Таким образом, этот результат может не соответствовать размеру, вычисленному путем сложения размеров элементов структуры.

```
buffer = calloc(100, sizeof(int));
```

Используя **sizeof**-операцию, можно избежать машинной зависимости, специфицируя в программе машинно-зависимые размеры данных. В примере используется операция **sizeof**, чтобы передать размер **int**, зависящий от машины, как аргумент функции, поименованной **calloc**. Значение, возвращаемое функцией, запоминается в буфер.

Мультипликативные операции

Мультипликативные операции выполняют операции умножения *, деления / и получения остатка от деления %. Операндами операции % должны быть целые числа. Операции умножения * и деления / выполняются над целыми и плавающими операндами. Типы первого и второго операндов могут отличаться.

Мультипликативные операции выполняют обычные арифметические преобразования операндов. Типом результата является тип операндов после преобразования.

Преобразования, выполненные посредством мультипликативных операций, не поддерживают ситуаций левого и правого переполнения. Информация теряется, если результат мультипликативной операции не может быть представлен в типе операндов после преобразования.

Умножение (*)

Операция умножения указывает на то, что ее оба операнда должны быть умножены.

Деление (/)

Операция деления указывает на то, что ее первый операнд делится на второй. Если две целые величины не делятся нацело, то результат усекается. Деление на **0** дает непредсказуемые результаты.

Остаток от деления (%)

Результатом операции является остаток от деления первого операнда на второй.

Аддитивные операции

Аддитивные операции выполняют сложение (+) и вычитание (-). Операндами могут быть целые и плавающие величины. В некоторых случаях аддитивные операции могут также выполняться на адресных величинах. На целых и плавающих операндах выполняются обычные арифметические преобразования. Типом результата является тип операндов после преобразования. Преобразования, выполняемые аддитивными операциями, не поддерживают левого и правого переполнения. Информация теряется, если результат аддитивной операции не может быть представлен типом операндов после преобразования.

Сложение (+)

Операция сложения специфицирует сложение двух операндов. Операнды могут быть целого или плавающего типов. Типы первого и второго операндов могут отличаться. Один операнд может быть указателем, а другой целой величиной. Когда целая величина складывается с указателем, то целая величина **i** преобразуется путем умножения ее на размер памяти, занимаемый величиной, адресуемой указателем. После преобразования целая величина представляет **i** позиций памяти, где каждая позиция имеет длину, специфицированную адресным типом. Когда преобразованная целая величина складывается с величиной указателя, то результатом является указатель, адресующий память, расположенную на **i** позиций дальше от исходного адреса. Новый указатель адресует тот же самый тип данных, что и исходный указатель.

Вычитание (-)

Операция вычитает второй операнд из первого. Операнды могут быть целого или плавающего типов. Типы первого и второго операндов могут отличаться. Допускается вычитание целого из указателя и вычитание двух указателей.

Когда целая величина вычитается из указателя, то перед выполнением операции производятся те же самые преобразования, что и при сложении целого с указателем. Результатом вычитания является указатель, адресующий память, расположенную на **i** позиций перед исходным адресом, где **i** целое, а каждая позиция — это длина типа, адресуемого указателем. Новый указатель адресует тот же самый тип данных, что и исходный указатель.

Один указатель может быть вычтен из другого, если они указывают на один и тот же тип данных. Разность между двумя указателями преобразуется к знаковой целой величине путем деления разности на длину типа, который адресуется указателями. Результат представляет число позиций памяти этого типа между двумя адресами.

Адресная арифметика

Аддитивные операции, применяемые к указателю и целому, имеют осмысленный результат, когда указатель адресует массив памяти, а целая величина представляет смещение адреса в пределах этого массива. Преобразование целой величины к адресному смещению предполагает, что в пределах смещения плотно расположены элементы одинакового размера. Это предположение справедливо для элементов массива. Массив определяется как набор величин одного и того же типа; его элементы расположены в смежных ячейках памяти.

Способ запоминания для любых типов, исключая элементы массива, не гарантирует плотного заполнения памяти.

Сложение и вычитание адресов, ссылающихся на любые величины, кроме элементов массива, дает непредсказуемый результат.

Аналогично, преобразования при вычитании двух указателей предполагают, что указатели ссылаются на величины одного и того же типа и что нет неиспользованной памяти между

элементами, расположенными в промежутке между адресами, заданными операндами.

Аддитивные операции между адресной и целой величинами на машинах с сегментной архитектурой (такие как 8086/8088) может быть неправильной в некоторых случаях.

Операции сдвига

Операции сдвига сдвигают свой первый операнд влево << или вправо >> на число позиций, специфицированных вторым операндом. Оба операнда должны быть целыми величинами. Обычные арифметические преобразования выполняются. Тип результата — это тип левого операнда после преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа, полученного после преобразования первого операнда. Если тип **unsigned**, то свободные левые биты устанавливаются в нуль.В противном случае они заполняются копией знакового бита. Результат операции сдвига не определен, если второй операнд отрицательный.

Преобразования, выполняемые операторами сдвига, не поддерживают левого и правого переполнения. Информация теряется, если результат сдвига не может быть представлен типом первого операнда после преобразования.

Операции отношений

Бинарные операции отношений сравнивают первый операнд со вторым и вырабатывают значение 1 (**true**) и 0 (**false**). Типом результата является **int**. Имеются следующие операции отношений.

Операции могут быть целого, плавающего или адресного типов. Типы первого и второго операндов могут различаться. Обычные арифметические преобразования выполняются над целыми и плавающими операндами.

Так как адрес данной величины произволен, то сравнение между адресами двух несвязанных величин, вообще говоря, не имеет смысла. Однако, сравнение между адресами различных элементов одного и того же массива может быть полезным, т.к. элементы массива хранятся в последовательном порядке. Адрес

первого элемента массива меньше чем адрес следующего элемента.

Адресная величина может быть сравнена на равенство или неравенство с константой 0. Указатель, имеющий значение 0, не указывает на область памяти. Он называется нулевым указателем. Значение указателя равно нулю, если оно таким явно задано путем присваивания или инициализации.

Побитовые операции

Побитовые операции выполняют побитовое И (**&**), включающее ИЛИ (**!**) и исключающее ИЛИ (**^**). Операнды побитовых операций должны быть целого типа, но их типы могут быть отличными. Обычные арифметические преобразования выполняются. Тип результата определяется типом операндов после преобразования.

Побитовое И (&)

Побитовое И сравнивает каждый бит своего первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае 0.

Побитовое включающее ИЛИ (!)

Побитовое включающее ИЛИ сравнивает каждый бит своего первого операнда с соответствующим битом второго операнда. Если любой из сравниваемых битов равен 1, то соответствующий бит результата устанавливается в 1. В противном случае оба бита равны 0 и соответствующий бит результата устанавливается в 0.

Побитовое исключающее ИЛИ (^)

Побитовое исключающее ИЛИ сравнивает каждый бит своего первого операнда с соответствующим битом второго операнда. Если один из сравниваемых битов равен 0, а второй бит равен 1, то соответствующий бит результата устанавливается в 1; в противном случае соответствующий бит результата устанавливается в 0.

Логические операции

Логические операции выполняют логическое И (**&&**) и логическое ИЛИ (**!!**). Операнды логических операций могут быть целого, плавающего или адресного типа. Типы первого и второго операндов могут быть различными. Операнды логических

выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, то второй операнд не вычисляется.

Логические операции не выполняют стандартные арифметические преобразования. Вместо этого они вычисляют каждый операнд с точки зрения его эквивалентности нулю. Указатель имеет значение 0, если это значение явно установлено путем присваивания или инициализации. Результатом логической операции является 0 или 1. Тип результата есть **int**.

Логическое И (&&)

Логическая операция И вырабатывает значение 1, если оба операнда имеют ненулевое значение. Если один из операндов равен 0, то результат также равен нулю. Если значение первого операнда равно нулю, то второй операнд не вычисляется.

Логическое ИЛИ (!!)

Логическая операция ИЛИ выполняет над своими операндами операцию включающего ИЛИ. Она вырабатывает значение 0, если оба операнда имеют значение 0; если какой-либо из операндов имеет ненулевое значение, то результат операции равен 1.

Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

Операция последовательного вычисления

Операция последовательного вычисления (**,**) вычисляет два своих операнда последовательно слева направо. Результат операции имеет значение и тип второго операнда. Типы операндов не ограничиваются. Преобразования не выполняются.

Условная операция

В Си есть одна тернарная операция — это условная операция **?:**. Она имеет следующее синтаксическое представление:

```
<operand 1>?<operand 2>:<operand 3>
```

Выражение **<operand 1>** вычисляется с точки зрения его эквивалентности нулю. Оно может быть целого, плавающего или адресного типа. Если **<operand 1>** имеет ненулевое значение, то вычисляется **<operand 2>** и результатом условной операции является значение выражения **<operand 2>**. Если **<operand 1>**

равен нулю, то вычисляется **<operand 3>** и результатом является значение выражения **<operand 3>**. Заметим, что вычисляется один из операндов **<operand 2>** или **<operand 3>**, но не оба.

Тип результата зависит от типов второго и третьего операндов следующим образом:

1. Если второй и третий операнды имеют целый или плавающий тип (их типы могут быть отличны), то выполняются обычные арифметические преобразования. Типом результата является тип операнда после преобразования.

2. Второй и третий операнды могут быть одного и того же структурного, совмещения или адресного типа. Тип результата будет тем же самым типом структуры, совмещения или адреса.

3. Один из второго или третьего операндов может быть указателем, а другой константным выражением со значением 0. Типом результата является адресный тип.

Операции присваивания

Операции присваивания в Си могут вычислять и присваивать значения в одной операции. Используя составные операции присваивания вместо двух отдельных операций, можно сократить код программы и улучшить ее эффективность.

Операциями присваивания являются следующие:

++

Унарный инкремент.

--

Унарный декремент.

=

Простое присваивание.

*=

Умножение с присваиванием.

/=

Деление с присваиванием.

%=

Остаток от деления с присваиванием.

+=

Сложение с присваиванием.

-=

Вычитание с присваиванием.

<<=

Сдвиг влево с присваиванием.

>>=

Сдвиг вправо с присваиванием.

&=

Побитовое И с присваиванием.

|=

Побитовое включающее ИЛИ с присваиванием.

^=

Побитовое исключающее ИЛИ с присваиванием.

При присваивании тип правого операнда преобразуется к типу левого операнда.

Lvalue-выражения

Операция присваивания означает, что значение правого операнда должно быть присвоено участку памяти, поименованному левым операндом. Поэтому левый операнд операции присваивания (или операнд унарного выражения присваивания) должен быть выражением, ссылающимся на участок памяти. Выражение, которое ссылается на участок памяти, называется **Lvalue**-выражением. Имя переменной является таким выражением: имя переменной указывает на участок памяти, а значением переменной является значение, находящееся в этой памяти.

Унарные инкремент и декремент

Унарная операция присваивания (**++** и **--**) инкрементирует или декрементирует свой операнд. Операнд должен быть целого, плавающего или адресного типа. В качестве операнда допустимо также **Lvalue**-выражение.Операнды целого или плавающего типа преобразуются путем сложения или вычитания целой 1. Тип результата соответствует типу операнда. Операнд адресного типа инкрементируется или декрементируется размером объекта, который он адресует. Инкрементированный указатель адресует

следующий объект, а декрементированный указатель — предыдущий.

Операции инкремента (++) или декремента (--) могут появляться перед или после своего операнда. Когда операция является префиксом своего операнда, то операнд инкрементируется или декрементируется и его новое значение является результатом вычисления выражения. Когда операция является постфиксом своего операнда, то непосредственным результатом выражения является значение операнда перед его инкрементированием или декрементированием. После этого результат используется в контексте, а операнд инкрементируется или декрементируется.

Простое присваивание

Операция простого присваивания (=) выполняет присваивание. Правый операнд присваивается левому операнду. При присваивании выполняются некоторые правила преобразования.

Составное присваивание

Операция составного присваивания состоит из простой операции присваивания, скомбинированной с другой бинарной операцией. В составном присваивании вначале выполняется операция, специфицированная аддитивным оператором, а затем результат присваивается левому операнду. Выражение составного присваивания, например, имеет вид:

```
<expression 1> += <expression 2>
```

и может быть понято как:

```
<expression 1> = <expression 1> + <expression 2>
```

Однако, выражение составного присваивания не эквивалентно расширенной версии, поскольку в выражении составного присваивания **<expression 1>** вычисляется только один раз, в то время как в расширенной версии оно вычисляется дважды: в операции сложения и в операции присваивания. Каждая операция составного присваивания выполняет преобразования, которые осуществляются соответствующей бинарной операций, и соответственно ограничивает типы своих операндов. Результатом операции составного присваивания является значение и тип левого операнда.

Старшинство и порядок выполнения

На старшинство и порядок выполнения операций Си влияют способы группирования и выполнения операндов в выражениях. Старшинство операций имеет смысл только при наличии нескольких операций, имеющих разные приоритеты. Выражения с более приоритетными операциями вычисляются первыми.

Старшинство операций уменьшается сверху вниз. Операции, расположенные в одной строке таблицы или объединенные в группу имеют одинаковое старшинство и одинаковый порядок выполнения.

Только операция последовательного вычисления (**,**) и логические операции И (**&&**) и ИЛИ (**!!**) обеспечивают определенный порядок вычисления операндов. Операция последовательного вычисления (**,**) обеспечивает преобразование своих операндов слева направо. (Заметим, что запятая, разделяющая аргументы в вызове функции, не является операцией последовательного вычисления и не обеспечивает таких гарантий.)

Логические операции также обеспечивают вычисление своих операндов слева направо. Однако логические операции вычисляют минимальное число операндов, необходимое для определения результата выражения. Таким образом, некоторые операнды выражения могут быть не вычислены. Например, в выражении «**x && y ++**» второй операнд «**y ++**» вычисляется только тогда, когда **x** есть **true** (не нуль). Так что **y** не инкрементируется, когда **x** есть **false** (нуль).

Побочные эффекты

Побочные эффекты — это изменения состояния машины, которые возникают в результате вычисления выражений. Они имеют место всякий раз, когда изменяется значение переменной. Любая операция присваивания вызывает побочный эффект, и любой вызов функции, который содержит операцию присваивания, имеет побочные эффекты.

Порядок получения побочных эффектов зависит от реализации, за исключением случаев, когда компилятор

обеспечивает определенный порядок вычислений. Например, побочный эффект имеет место в следующем вызове функции:

add (i + 1, i = j +2)

Аргументы вызова функции могут быть вычислены в любом порядке. Выражение «**i + 1**» может быть вычислено перед «**i=j+2**», или наоборот, с различным результатом в каждом случае.

Унарные операции инкремента и декремента включают присваивание и могут быть причиной побочных эффектов, как это показано в следующем примере:

```
d=0
a=b++=c++=d++;
```

Значение **a** непредсказуемо. Значение **d** (инициализируется нулем), могло быть присвоено **c**, затем **b** и затем **a**, прежде чем любая из переменных была бы инкрементирована. В этом случае **a** должно было бы быть эквивалентно нулю.

Второй способ вычисления этого выражения начинается вычислением операнда **c++=d++**. Значение **d** (инициализированное нулем) присваивается **c**, а затем **d** и **c** инкрементируются. Затем значение **c**, которое теперь равно 1, присваивается **b** и **b** инкрементируется.

Наконец, инкрементированное значение **b** присваивается **a**. В этом случае окончательное значение **a** равно 2.

Так как язык Си не определяет порядок изменения состояний машины (побочных эффектов) при вычислениях, то оба эти метода вычисления корректны и могут быть выполнены. Операторы, которые зависят от частностей порядка вычисления побочных эффектов, выдают непереносимый и неясный код.

Преобразования типов

Преобразование типов имеет место, когда тип значения, которое присваивается переменной, отличается от типа переменной. Преобразование типов выполняется, когда операция перед вычислением преобразует тип своего операнда или операндов и когда преобразуется значение, посылаемое как аргумент функции.

Преобразование типов при присваивании

В операциях присваивания тип значения, которое присваивается, преобразуется к типу переменной, получающей это значение. В Си допускаются преобразования при присваивании между целыми и плавающими типами, даже в случаях, когда преобразование влечет за собой потерю информации.

Знаковое целое преобразуется к короткому знаковому целому (**short signed int**) посредством усечения старших битов.

Знаковое целое преобразуется к длинному знаковому целому (**long signed int**) путем размножения знака влево. Преобразование знаковых целых к плавающим величинам происходит без потери информации, за исключением потери некоторой точности, когда преобразуются величины **long** в **float**. При преобразовании знакового целого к беззнаковому целому (**unsigned int**), знаковое целое преобразуется к размеру беззнакового целого и результат интерпретируется как беззнаковая величина.

Тип **unsigned int** эквивалентен или **unsigned short**, или **unsigned long** типам в зависимости от оборудования. Преобразование из **unsigned int** производятся как для **unsigned short** или **unsigned long** в зависимости от того, что подходит.

Преобразование плавающих типов

Величины **float** преобразуются к **double**, не меняясь в значении. Величины **double**, преобразованные к **float**, представляются точно, если возможно. Если значение слишком велико для **float**, то точность теряется.

Плавающие величины преобразуются к целым типа **long**. Преобразование к другим целым типам выполняется как для **long**. Дробная часть плавающей величины отбрасывается при преобразовании к **long**; если результат слишком велик для **long**, то результат преобразования неопределен.

Преобразование адресных типов

Указатель на величину одного типа может быть преобразован к указателю на величину другого типа. Результат может быть, однако, неопределенным из-за отличия в требованиях к выравниванию и размерам памяти.

В некоторых реализациях имеются специальные ключевые слова **near**, **far**, **huge**, модифицирующие размер указателей в программах. Указатель может быть преобразован к указателю другого размера; путь преобразования зависит от реализации.

Значение указателя может быть преобразовано к целой величине. Путь преобразования зависит от размера указателя.

Целый тип может быть преобразован к адресному типу. Если целый тип того же самого размера, что и адресный, то производится простое преобразование к виду указателя (беззнакового целого). Если размер целого типа отличен от размера адресного типа, то целый тип вначале преобразуется к размеру указателя. Затем полученное значение представляется как указатель.

Если поддерживаются специальные ключевые слова **near**, **far**, **huge**, то может быть сделано неявное преобразование адресных величин. В частности, компилятор может по умолчанию создавать указатели определенного размера и производить преобразования получаемых адресных величин, если в программе не появится **forward** объявление, переопределяющее это умолчание.

Преобразования других типов

Из определения типа **enum** следует, что величины **enum** являются величинами типа **int**. Поэтому преобразования в и из типа **enum** осуществляется как для **int** типов. Тип **int** эквивалентен типам **short** или **long** в зависимости от реализации.

Не допустимы преобразования объектов типа структур и совмещений.

Тип **void** не имеет значения по определению. Поэтому он не может быть преобразован к любому другому типу, но любая величина может быть преобразована к типу **void** путем присваивания. Тем не менее, величина может быть явно преобразована **cast** операцией к **void**.

Преобразования type-cast

Явное преобразование типа может быть сделано посредством **type-cast**. Преобразования **type-cast** имеют следующую синтаксическую форму:

```
(<type-name>)<operand>
```

где **<type-name>** специфицирует особенности типа, а **<operand>** является величиной, которая должна быть преобразована к специфицированному типу.

Преобразование операнда осуществляется в процессе присвоения его переменной типа **<type-name>**. Правила преобразования для операции присваивания допустимы для преобразований **type-cast** полностью. Имя типа **void** может быть использовано в операции **cast**, но результирующее выражение не может быть присвоено любому объекту.

Преобразования, выполняемые операциями

Преобразования, выполняемые операциями Си, зависят от самих операций и от типа операнда или операндов. Многие операции выполняют обычные арифметические преобразования.

Си разрешает некоторую арифметику с указателями. В адресной арифметике целые величины преобразуются к определенным адресам памяти.

Преобразования при вызовах функций

Тип преобразования, выполняемый над аргументами в вызове функции, зависит от того, было ли **forward** объявление, определяющее типы аргументов для вызываемой функции.

Если **forward** объявление было и оно включает определение типов аргументов, то компилятор осуществляет контроль типов.

Если **forward** объявления не было, или если в **forward** объявлении опущен список типов аргументов, то над аргументами вызываемой функции производятся только обычные арифметические преобразования. Преобразования производятся независимо для каждого аргумента вызова. Смысл этих преобразований сводится к тому, что величины типа **float** преобразуются к **double**, величины типов **char** и **short** преобразуются к **int**, величины типов **unsigned char** и **unsigned short** преобразуются к **unsigned int**. Если поддерживаются специальные ключевые слова **near**, **far**, **huge**, то могут быть также сделаны неявные преобразования адресных величин, посылаемых в функцию. Эти неявные преобразования могут быть переопределены заданием в **forward** объявлении списка типов аргументов, что позволит компилятору выполнить контроль типов.

Операторы

Операторы Си управляют процессом выполнения программы. В Си, как и в других языках программирования, имеются условные операторы, операторы цикла, выбора, передачи управления и т.д. Ниже представлен список операторов в алфавитном порядке:

```
break
<compound>
continue
do
<expression>
for
goto
if
<null>
return
switch
while
```

Операторы Си состоят из ключевых слов, выражений и других операторов. В операторах Си допустимы следующие ключевые слова:

```
break
default
for
return
case
do
goto
switch
continue
else
if
while
```

Операторами, допустимыми внутри операторов Си, могут быть любые операторы. Оператор, который является компонентом другого оператора, называется «телом» включающего оператора.Часто оператор-тело является составным оператором, состоящим из одного или более операторов.

Составной оператор ограничивается фигурными скобками. Все другие операторы Си заканчиваются точкой с запятой **;**.

Любой из операторов Си может быть спереди помечен меткой, состоящей из имени и двоеточия. Операторные метки опознаются только оператором **goto**.

Порядок выполнения программы Си совпадает с порядком расположения операторов в тексте программы, за исключением тех случаев, когда оператор явно передает управление в другую часть программ.

Оператор break

Синтаксис:

```
break;
```

Оператор **break** прерывает выполнение операторов **do**, **for**, **switch** или **while**, в которых он появляется. Управление передается оператору, следующему за прерванным. Появление оператора **break** вне операторов **do**, **for**, **switch**, **while** приводит к ошибке.

Внутри вложенных операторов оператор **break** завершает только операторы **do**, **for**, **switch** или **while**. Чтобы передать управление вне вложенной структуры, могут быть использованы операторы **return** и **goto**.

Составной оператор

Синтаксис:

```
[<declaration>]
.
.
.
<statement>
[<statement>]
.
.
.
```

Действия при выполнении составного оператора состоят в том, что выполнение его операторов осуществляется в порядке их появления, за исключением случаев, когда очередной оператор явно передает управление в другое место.

Помеченные операторы

Подобно другим операторам Си, любой оператор в составном операторе может быть помечен. Поэтому передача управления внутрь составного оператора возможна. Однако, передачи управления внутрь составного оператора опасны, когда составной оператор содержит объявления, которые инициализируют переменные. Объявления в составном операторе предшествуют выполняемым операторам, так что передача управления непосредственно на выполняемый оператор внутри составного оператора минует инициализацию. Результат будет непредсказуем.

Оператор continue

Синтаксис:

```
continue;
```

Оператор **continue** передает управление на следующую итерацию в операторах цикла **do**, **for**, **while**, в которых он может появиться. Оставшиеся операторы в теле вышеперечисленных циклов при этом не выполняются. Внутри **do** или **while** циклов следующая итерация начинается с перевычисления выражения **do** или **while** операторов. Для оператора **for** следующая итерация начинается с выражения цикла оператора **for**.

Оператор do

Синтаксис:

```
do
<statement>
while (<expression>);
```

Тело оператора **do** выполняется один или несколько раз до тех пор, пока выражение **<expression>** станет ложным (равным нулю). Вначале выполняется оператор **<statement>** тела, затем вычисляется выражение **<expression>**. Если выражение ложно, то оператор **do** завершается и управление передается следующему оператору в программе. Если выражение истинно (не равно нулю), то тело оператора выполняется снова и снова проверяется выражение. Выполнение тела оператора продолжается до тех пор, пока выражение не станет ложным. Оператор **do** может также завершить выполнение при выполнении операторов **break**, **goto** или **return** внутри тела оператора **do**.

Оператор-выражение

Синтаксис:

```
expression;
```

Выражение **<expression>** вычисляется в соответствии с правилами «Выражения и присваивания».

В Си присваивания являются выражениями. Значением выражения является значение, которое присваивается.

Оператор for

Синтаксис:

```
for ([<init-expression>];[<cond-expression>];[<loop-exp>])
statement
```

Тело оператора **for** выполняется нуль и более раз, до тех пор, пока условное выражение **<cond-expression>** не станет ложным. Выражения инициализации **<init-expression>** и цикла **<loop-expression>** могут быть использованы для инициализации и модификации величин во время выполнения оператора **for**.

Первым шагом при выполнении оператора **for** является вычисление выражения инициализации, если оно имеется. Затем вычисление условного выражения с тремя возможными результатами:

1. Если условное выражение истинно (не равно нулю), то выполняется тело оператора. Затем вычисляется выражение цикла (если оно есть). Процесс повторяется снова с вычислением условного выражения.

2. Если условное выражение опущено, то его значение принимается за истину и процесс выполнения продолжается, как показано выше. В этом случае оператор **for** может завершиться только при выполнении в теле оператора операторов **break**, **goto**, **return**.

3. Если условное выражение ложно, то выполнение оператора **for** заканчивается и управление передается следующему оператору в программе.

Оператор **for** может завершиться при выполнении операторов **break**, **return**, **goto** в теле оператора.

Оператор **goto** передает управление непосредственно на оператор, помеченный **<name>**. Помеченный оператор выполняется сразу после выполнения оператора **goto**. Если

оператор с данной меткой отсутствует или существует более одного оператора, помеченных одной и той же меткой, то это приводит к ошибочному результату. Метка оператора имеет отношение только к оператору **goto**. Если помеченный оператор встречается в любом другом контексте, то он выполняется без учета метки.

Пример:

```
if (errorcode>0)
goto exit;
.
.
.
exit:return (errorcode);
```

В примере оператор **goto** передает управление на оператор, помеченный меткой **exit**, когда происходит ошибка.

Формат меток

Метка — это простой идентификатор. Каждая метка должна быть отлична от других меток в той же самой функции.

Оператор if

Синтаксис:

```
if (<expression>)
<statement 1>
[else
<statement 2>]
```

Тело оператора **if** выполняется селективно, в зависимости от значения выражения **<expression>**. Сначала вычисляется выражение. Если значение выражения истина (не нуль), то выполняется оператор **<statement 1>**. Если выражение ложно, то выполняется оператор **<statement 2>**, непосредственно следующий за ключевым словом **else**. Если выражение **<expression>** ложно и предложение **else ...** опущено, то управление передается на выполнение оператора, следующего за оператором **if**.

Пример:

```
if (i>0)
y=x/i;
else
```

```
x=i;
y=f(x);
```

Вложения

Си не поддерживает оператор **else if**, но тот же самый эффект достигается посредством сложенных операторов **if**. Оператор **if** может быть вложен в предложение **if** или предложение **else** другого оператора **if**. Когда операторы **if** вкладываются, то используются фигурные скобки, чтобы сгруппировать составные операторы, которые проясняют ситуацию.

Если фигурные скобки отсутствуют, то компилятор может принять неверное решение, сочетая каждое **else** с более близким **if**, у которого отсутствует **else**.

Пример:

```
/****** example 1 ******/
if (i>0) /* without braces */
if (j>i)
x=j;
else
x=i;
/****** example 2 ******/
if (i>0)  /* with braces */
if (j>1)
x=j;

else
x=i;
```

Оператор null

Синтаксис:

```
;
```

Оператор **null** — это оператор, состоящий только из точки с запятой. Он может появиться в любом месте, где требуется оператор. Когда выполняется оператор **null**, ничего не происходит.

Пример:

```
for (i=0; i<10; line [i++]=0)
;
```

Такие операторы, как **do**, **for**, **if**, **while**, требуют, чтобы в теле оператора был хотя бы один оператор. Оператор **null** удовлетворяет требованиям синтаксиса в случаях, когда не требуется тела оператора. В приведенном примере третье выражение оператора **for** инициализирует первые 10 элементов массива **line** нулем. Тело оператора включает оператор **null**, т.к. нет необходимости в других операторах.

Помеченный оператор null

Оператор **null**, подобно любому другому Си оператору, может быть помечен меткой. Чтобы пометить объект, который не является оператором, такой как закрывающаяся фигурная скобка составного оператора, можно вставить перед объектом помеченный оператор **null**.

Оператор return

Синтаксис:

```
return [<expression>];
```

Оператор **return** заканчивает выполнение функции, в которой он появляется, и возвращает управление в вызывающую функцию. Управление передается в вызывающую функцию в точку, непосредственно следующую за вызовом. Значение выражения **<expression>**, если оно есть, возвращается в вызывающую функцию. Если выражение **<expression>** опущено, то возвращаемая функцией величина не определена.

Пример:

```
main ()
void draw (int,int); long sq (int);
.
.
.
y=sq (x);
draw (x,y);
.
.
.

long sq (x)
int x;
```

```
return (x*x);

void draw (x,y)
int x,y;


.
.
.
return;
```

Функция **main** вызывает две функции: **sq** и **draw**. Функция **sq** возвращает значение **x*x** в **main**. Величина возврата присваивается переменной **y**. Функция **draw** объявляется как функция **void** и не возвращает значения. Попытка присвоить возвращаемое значение функции **draw** привело бы к ошибке.

Выражение **<expression>** оператора **return** заключено в скобки, как показано в примере. Язык не требует скобок.

Отсутствие оператора return

Если оператор **return** не появился в определении функции, то управление автоматически передается в вызывающую функцию после выполнения последнего оператора в вызванной функции. Значение возврата вызванной функции при этом не определено. Если значение возврата не требуется, то функция должна быть объявлена с типом возврата **void**.

Оператор switch

Синтаксис:

```
switch (<expression>)
[<declaration>]
.
.
.
[case <constant-expression>:]
.
.
.
[<statement>]
.
.
.
```

```
[default:
<statement>]
[case <constant-expression>:]
.
.
.
[<statement>]
.
.
.
```

Оператор **switch** передает управление одному из операторов **<statement>** своего тела. Оператор, получающий управление, — это тот оператор, чье **case**-константное выражение **<constant-expression>** равно значению **switch**-выражения **<expression>** в круглых скобках.

Выполнение тела оператора начинается с выбранного оператора и продолжается до конца тела или до тех пор, пока очередной оператор **<statement>** передает управление за пределы тела.

Оператор **default** выполнится, если **case**-константное выражение **<constant-expression>** не равно значению **switch**-выражения **<expression>**. Если **default**-оператор опущен, а соответствующий **case** не найден, то выполняемый оператор в теле **switch** отсутствует. **Switch**-выражение **<expression>** — это целая величина размера **int** или короче. Оно может быть также величиной типа **enum**. Если **<expression>** короче чем **int**, оно расширяется до **int**.

Каждое **case**-константное выражение **<constant-expression>** преобразуется к типу **switch**-выражения. Значение каждого **case**-константного выражения должно быть уникальным внутри тела оператора.

Case и **default** метки в теле оператора **switch** существенны только при начальной проверке, когда определяется стартовая точка для выполнения тела оператора. Все операторы появляющиеся между стартовым оператором и концом тела, выполняются, не обращая внимания на свои метки, если какой-то из операторов не передает управления из тела оператора **switch**.

В заголовке составного оператора, формирующего тело оператора **switch**, могут появиться объявления, но инициализаторы, включенные в объявления, не будут выполнены. Назначение оператора **switch** состоит в том, чтобы передать управление непосредственно на выполняемый оператор внутри тела, обойдя строки, которые содержат инициализацию.

Пример:

```
/***.....* example 1 *.....***/
switch (c)
case 'A':
capa++;
case 'a':
lettera++;
default:
total++;
```

Множественные метки

Оператор тела **switch** может быть помечен множественными метками, как показано в следующем примере:

```
case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
case 'f': hexcvt (c);
```

Хотя любой оператор внутри тела оператора **switch** может быть помечен, однако не требуется оператора, чтобы появилась метка. Операторы без меток могут быть смешаны с помеченными операторами. Следует помнить, однако, что если **switch** оператор передал управление одному из операторов своего тела, то все следующие за ним операторы в блоке выполняются, не обращая внимания на свои метки.

Оператор while

Синтаксис:

```
while (<expression>)
<statement>
```

Тело оператора **while** выполняется нуль или более раз до тех пор, пока выражение **<expression>** станет ложным (равным нулю). Вначале вычисляется выражение **<expression>**. Если **<expression>**

изначально ложно, то тело оператора **while** не выполняется и управление передается на следующий оператор программы. Если **<expression>** является истиной (не нуль), то выполняется тело оператора. Перед каждым следующим выполнением тела оператора **<expression>** перевычисляется. Повторение выполнения тела оператора происходит до тех пор, пока **<expression>** остается истинным. Оператор **while** может также завершиться при выполнении операторов **break**, **goto**, **return** внутри тела **while**.

Пример:

```
while (i>=0)
string1 [i] = string2 [i];
i--;
```

В вышеприведенном примере копируются символы из **string2** в **string1**. Если **i** больше или равно нулю, то **string2[i]** присваивается индексной переменной **string1[i]** и **i** декрементируется. Когда **i** становится меньше нуля, то выполнение оператора **while** завершается.

Функции

Функция — это независимая совокупность объявлений и операторов, обычно предназначенная для выполнения определенной задачи. Программы на Си состоят по крайней мере из одной функции **main**, но могут содержать и больше функций.

Определение функции специфицирует имя функции, ее формальные параметры, объявления и операторы, которые определяют ее действия. В определении функции может быть задан также тип возврата и ее класс памяти.

В объявлении задается имя, тип возврата и класс памяти функции, чье явное определение произведено в другой части программы. В объявлении функции могут быть также специфицированы число и типы аргументов функции. Это позволяет компилятору сравнить типы действительных аргументов и формальных параметров функции. Объявления не обязательны для функций, возвращающих величины типа **int**.

Чтобы обеспечить корректное обращение при других типах возвратов, необходимо объявить функцию перед ее вызовом.

Вызов функции передает управление из вызывающей функции к вызванной. Действительные аргументы, если они есть, передаются по значению в вызванную функцию. При выполнении оператора return в вызванной функции управление и, возможно, значение возврата передаются в вызывающую функцию.

Определение функции

Определение функции специфицирует имя, формальные параметры и тело функции. Оно может также определять тип возврата и класс памяти функции. Синтаксис определения функции следующий:

```
[<sc-specifier>][<type-specifier>]<declarator>
([<parameter-list>])
[<parameter-declarations>]
<function-body>
```

Спецификатор класса памяти **<sc-specifier>** задает класс памяти функции, который может быть или **static** или **extern**. Спецификатор типа **<type-specifier>** и декларатор **<declaration>** специфицируют тип возврата и имя функции. Список параметров **<parameter-list>** — это список (возможно пустой) формальных параметров, которые используются функцией. Объявления параметров **<parameter-declarations>** задают типы формальных параметров. Тело функции **<function-body>** — это составной оператор, содержащий объявления локальных переменных и операторы.

Класс памяти

Спецификатор класса памяти в определении функции определяет функцию как **static** или **extern**. Функция с классом памяти **static** видима только в том исходном файле, в котором она определена. Все другие функции с классом памяти **extern**, заданным явно или неявно, видимы во всех исходных файлах, которые образуют программу.

Если спецификатор класса памяти опускается в определении функции, то подразумевается класс памяти **extern**. Спецификатор класса памяти **extern** может быть явно задан в определении функции, но этого не требуется.

Спецификатор класса памяти требуется при определении функции только в одном случае, когда функция объявляется

где-нибудь в другом месте в том же самом исходном файле с спецификатором класса памяти **static**. Спецификатор класса памяти **static** может быть также использован, когда определяемая функция предварительно объявлена в том же самом исходном файле без спецификатора класса памяти. Как правило, функция, объявленная без спецификатора класса памяти, подразумевает класс **extern**. Однако, если определение функции явно специфицирует класс **static**, то функции дается класс **static**.

Тип возврата

Тип возврата функции определяет размер и тип возвращаемого значения. Объявление типа имеет следующий синтаксис:

```
[<type-specifier>] <declarator>
```

где спецификатор типа **<type-specifier>** вместе с декларатором **<declarator>** определяет тип возврата и имя функции. Если **<type-specifier>** не задан, то подразумевается, что тип возврата **int**. Спецификатор типа может специфицировать основной, структурный и совмещающий типы. Декларатор состоит из идентификатора функции, возможно модифицированного с целью объявления адресного типа. Функции не могут возвращать массивов или функций, но они могут возвращать указатели на любой тип, включая массивы и функции. Тип возврата, задаваемый в определении функции, должен соответствовать типам возвратов, заданных в объявлениях этой функции, сделанных где-то в программе. Функции с типом возврата **int** могут не объявляться перед вызовом. Функции с другими типами возвратов не могут быть вызваны прежде, чем они будут определены или объявлены.

Тип значения возврата функции используется только тогда, когда функция возвращает значение, которое вырабатывается, если выполняется оператор **return**, содержащий выражение. Выражение вычисляется, преобразуется к типу возврата, если это необходимо, и возвращается в точку вызова. Если оператор **return** не выполняется или если выполняемый оператор **return** не содержит выражения, то значение возврата функции не определено. Если в этом случае вызывающая функция ожидает значение возврата, то поведение программы также не определено.

Формальные параметры

Формальные параметры — это переменные, которые принимают значения, переданные функции от функционального вызова. Формальные параметры объявляются в списке параметров в начале описания функции. Список параметров определяет имена параметров и порядок, в котором они принимают значения при вызове функции.

Тело функции

Тело функции — это просто составной оператор. Составной оператор содержит операторы, которые определяют действия функции, и может также содержать объявления переменных, используемых в этих операторах. Все переменные, объявленные в теле функции, имеют тип памяти **auto**, если они не объявлены иначе. Когда вызывается функция, то создается память для локальных переменных и производится их инициализация (если она задана). Управление передается первому оператору составного оператора и начинается процесс выполнения, который продолжается до тех пор, пока не встретится оператор **return** или конец тела функции. Управление при этом возвращается в точку вызова.

Если функция возвращает значение, то должен быть выполнен оператор **return**, содержащий выражение.

Значение возврата не определено, если не выполнен оператор **return** или если в оператор **return** не было включено выражение.

Объявления функции

Объявление функции определяет имя, тип возврата и класс памяти данной функции и может задавать тип некоторых или всех аргументов функции.

Функции могут быть объявлены неявно или в **forward**-объявлениями. Тип возврата функции, объявленный или неявно или в **forward**-объявлении, должен соответствовать типу возврата в определении функции. Неявное объявление имеет место всякий раз, когда функция вызывается без предварительного объявления или определения. Си-компилятор неявно объявляет вызываемую функцию с типом возврата **int**. По умолчанию функция объявляется с классом памяти **extern**. Определение функции может переопределить класс памяти на

static, обеспечив себе появление ниже объявлений в том же самом исходном файле. **Forward**-объявление функции устанавливает ее атрибуты, позволяя вызывать объявленную функцию перед ее определением или из другого исходного файла.

Если спецификатор класса памяти **static** задается в **forward**-объявлении, то функция имеет класс **static**. Поэтому определение функции должно быть также специфицировано классом памяти **static**.

Если задан спецификатор класса памяти **extern** или спецификатор опущен, то функция имеет класс памяти **extern**. Однако определение функции может переопределить класс памяти на **static**, обеспечив себе появление ниже объявлений в том же самом исходном файле

Forward-объявление имеет важные различные применения. Они задают тип возврата для функций, которые возвращают любой тип значений, за исключением **int**. (Функции, которые возвращают значение **int**, могут также иметь **forward**-объявления, но делать это не требуется).

Если функция с типом возврата не **int** вызывается перед ее определением или объявлением, то результат неопределен. **Forward**-объявления могут быть использованы для задания типов аргументов, ожидаемых в функциональном вызове.

Список типов аргументов **forward**-объявления задает тип и число предполагаемых аргументов. (Число аргументов может меняться). Список типов аргументов — это список имен типов, соответствующих списку выражений в функциональном вызове.

Если список типов аргументов не задан, то не производится контроль типов. Несоответствие типов между действительными аргументами и формальными параметрами разрешено.

Вызовы функций

Вызов функции — это выражение, которое передает управление и фактические аргументы (если они есть) функции. Вызов функции имеет следующее синтаксическое представление:

```
<expression> ([<expression-list>])
```

Выражение **<expression>** вычисляется как адрес функции. Список выражение **<expression-list>**, в котором выражения следуют через запятую, представляет список фактических

аргументов, посылаемых функции. Список выражений может быть пустым.

При выполнении вызова функции происходит замена формальных аргументов на фактические. Перед заменой каждый из фактических аргументов вычисляется. Первый фактический аргумент соответствует первому формальному аргументу, второй — второму и т.д.

Вызванная функция работает с копией действительных аргументов, поэтому любое изменение, сделанное функцией с аргументами, не отразится на оригинальных величинах, с которых была сделана копия.

Передача управления осуществляется на первый оператор функции. Выполнение оператора **return** в теле функции возвращает управление и, возможно, значение возврата в вызывающую функцию. Если оператор **return** не выполнен, то управление возвращается после выполнения последнего оператора тела функции. При этом величина возврата не определена.

Важно: Выражения в списке аргументов вызова функции могут выполняться в любом порядке, так что выражения с побочными эффектами могут дать непредсказуемые результаты. Компилятор только гарантирует, что все побочные эффекты будут вычислены перед передачей управления в вызываемую функцию.

Выражение перед скобками должно быть преобразовано к адресу функции. Это означает, что функция может быть вызвана через любое выражение типа указателя на функцию. Это позволяет вызывать функцию в той же самой манере, что и объявлять. Это необходимо при вызове функции через указатель. Например, предположим, что указатель на функцию объявлен как следующий:

```
int (* fpointer)(int,int);
```

Идентификатор **fpointer** указывает на функцию с двумя аргументами типа **int** и возвращающую значение типа **int**. Вызов функции в этом случае будет выглядеть следующим образом:

```
(* fpointer)(3,4);
```

Здесь используется операция разадресации (*), чтобы получить адрес функции, на которую ссылается указатель **fpointer**. Адрес функции затем используется для ее вызова.

Фактические аргументы

Фактические аргументы могут быть любой величиной основного, структурного, совмещающего или адресного типов. Хотя массивы и функции не могут быть переданы как параметры, но указатели на эти объекты могут передаваться. Все фактические аргументы передаются по значению. Копия фактических аргументов присваивается соответствующим формальным параметрам. Функция использует эти копии, не влияя на переменные, с которых копия была сделана.

Путь доступа из функции к значениям оригинальных переменных обеспечивают указатели. Т.к. указатель на переменную содержит адрес переменной, то функция может использовать этот адрес для доступа к значению переменной. Аргументы-указатели обеспечивают доступ из функции к массивам и функциям, которые запрещено передавать как аргументы.

Тип каждого формального параметра также подвергается обычным арифметическим преобразованиям. Преобразованный тип каждого формального параметра определяет, каким образом интерпретируются аргументы в стеке. Если тип формального параметра не соответствует типу фактического параметра, то данные в стеке могут быть проинтерпретированы неверно.

Важно: Несоответствие типов формальных и фактических параметров может произвести серию ошибок, особенно когда несоответствие влечет за собой отличия в размерах. Нужно иметь в виду, что эти ошибки не выявляются, если не задан список типов аргументов в **forward**-объявлении функции.

Вызовы с переменным числом аргументов

Чтобы вызвать функцию с переменным числом аргументов, в вызове функции просто задается любое число аргументов. В **forward**-объявлении (если оно есть) переменное число аргументов специфицируется записью запятой с последующим многоточием (**,...**) в конце списка типов аргументов. Каждому имени типа, специфицированному в списке типов аргументов, соответствует один фактический аргумент в вызове функции. Если задано только многоточие (без имен типов), то это значит, что нет аргументов, которые обязательно требуются при вызове функции.

Аналогично, список аргументов в определении функции может также заканчиваться запятой с последующим многоточием (**,...**), что подразумевает переменное число аргументов.

Если список аргументов содержит только многоточие (**...**), то число аргументов переменно или равно нулю.

Важно: Для поддержки совместимости с предыдущими версиями компилятор воспринимает для обозначения переменного числа аргументов символ запятой без последующего многоточия в конце списка типов аргументов или списка параметров. Так же может быть использована отдельная запятая вместо многоточия для объявления и определения функций, требующих нуль или более аргументов. Использование запятой поддерживается только для совместимости. Желательно использовать в новой версии многоточие.

Все аргументы, заданные в версии функции, размещаются в стеке. Количество формальных параметров, объявленных для функции, определяет число аргументов, которые берутся из стека и присваиваются формальным параметрам. Программист отвечает за выбор лишних аргументов из стека и за то, сколько аргументов находится в стеке. Смотрите в системной документации информацию о макросах, которые могут быть использованы для управления переменным числом аргументов.

Рекурсивные вызовы

Любая функция в Си-программе может быть вызвана рекурсивно. Для этого функция вызывает саму себя. Компилятор Си допускает любое число рекурсивных вызовов функции. При каждом вызове формальных параметров и переменных класса памяти **auto** и **register** захватывается новая память, так что их значения из предшествующих незавершенных вызовов не перезаписываются. Предшествующие параметры недоступны в других версиях функции, исключая версию, в которой они были созданы.

Заметим, что переменные, объявленные как глобальные, не требуют новой памяти при каждом рекурсивном вызове. Их память сохраняется на все время жизни программы.

Язык программирования C++

Введение

Автором языка C++ является Бьерн Страуструп, сотрудник известной фирмы AT&T. C++ (а точнее, его предшественник, Си with classes) был создан под влиянием Simula (надо сказать, что этот язык программирования появился еще в 1967 году). Собственно, к тому моменту (когда появлялся C++), Си уже заработал себе популярность; обычно, его очень уважают за возможность использовать возможности конкретной архитектуры, при этом все еще используя язык относительно высокого уровня. Правда, обычно, именно за это его и не любят. Конечно же, при этом терялись (или извращались) некоторые положения построения программ; например, в Си фактически отсутствует возможность создавать модульные программы. Нет, их конечно же создают, но при помощи некоторых «костылей» в виде использования директив препроцессора — такой модульности, как, например в Modula-2 или Ada, в Си нет. Кстати сказать, этого нет до сих пор и в C++.

С самого начала подчеркивалось то, что C++ — развитие языка Си, возможно, некоторый его диалект. Об этом говорит тот факт, что первым компилятором (существующим до сих пор) являлся **cfront**, который занимался тем, что переводил исходный текст на C++ в исходный текст на Си. Это мнение бытует до сих пор и, надо сказать, до сих пор оно является небезосновательным; тем не менее, Си и C++ — разные языки программирования.

Основное отличие C++, когда он только появлялся, была явная поддержка объектно-ориентированного подхода к программированию. Надо понимать, что программировать с использованием ООП и ООА можно где угодно, даже если инструментарий явно его не поддерживает; в качестве примера можно взять библиотеку пользовательского интерфейса GTK+, которая написана на «чистом» Си и использованием принципов объектно-ориентированного «дизайна». Введение в язык программирования средств для поддержки ООП означает то, что на стадии компиляции (а не на стадии выполнения программы)

будет производится проверка совместимости типов, наличия методов и т.п. В принципе, это достаточно удобно.

Опять же, точно так же как и Си не является в чистом виде языком программирования высокого уровня (из-за того, что позволяет выполнить слишком много трюков), C++, строго говоря, не является объектно-ориентированным языком программирования. Мешают этому такие его особенности, как наличие виртуальных функций (потому что при разговоре о полиморфизме никто никогда не оговаривает того, что некоторые методы будут участвовать в нем, а некоторые — нет), присутствие до сих пор функции **main()** и т.п. Кроме того, в C++ нет таких сущностей, как, например, метаклассы (хотя они, наверное, не так уж сильно и нужны) и интерфейсы (вместо них используется множественное наследование). Тем не менее, C++ на текущий момент один из самых популярных (если не самый популярный) язык программирования. Почему? Да потому что все эти «уродства» позволяют в итоге написать программу с использованием объектно-ориентированных подходов (а программы, которые этого требуют, обычно очень большие) и при этом достаточно «быструю». Этому способствует именно наличие виртуальных функций (т.е., то, что программист сам разделяет еще во время проектирования, где ему понадобится полиморфизм, а где будет достаточно объединить некоторые функции в группу по некоторому признаку), обязательное наличие оптимизатора и прочее. Все это позволяет при грамотном использовании все-таки написать работающую программу. Правда, достаточно часто встречаются примеры неграмотного использования, в результате чего выбор C++ для реализации проекта превращается в пытку для программистов и руководства.

На текущий момент на язык программирования C++ существует стандарт, в который включена, помимо прочего, стандартная библиотека шаблонов. О шаблонах разговор особый, формально они тоже являются «костылями» для того, чтобы сделать, так сказать, полиморфизм на стадии компиляции (и в этом качестве очень полезны), а вот библиотека — бесспорно правильное со всех сторон новшество. Наличие, наконец-то, стандартных способов переносимо (имеется в виду, с компилятора на компилятор) отсортировать список (кроме написания всех соответствующих подпрограмм и структур данных

самостоятельно) очень сильно облегчило жизнь программиста. Правда, до сих пор очень много людей плохо себе представляют как устроена STL и как ей пользоваться (кроме **std::cin** и **std::cout**).

Важной вехой в развитии программирования явилось создание и широкое распространение языка С++. Этот язык, сохранив средства ставшего общепризнанным стандартом для написания системных и прикладных программ языка Си (процедурно-ориентированный язык), ввел в практику программирования возможности нового технологического подхода к разработке программного обеспечения, получившего название «объектно-ориентированное программирование».

Язык программирования **С++ —** это **С***, расширенный введением классов, **inline**-функций, перегруженных операций, перегруженных имен функций, константных типов, ссылок, операций управления свободной памятью, проверки параметров функций.

Внедрение в практику написания программ объектно-ориентированной парадигмы дает развитие новых областей информатики, значительное повышение уровня технологичности создаваемых программных средств, сокращение затрат на разработку и сопровождение программ, их повторное использование, вовлечение в процесс расширения интеллектуальных возможностей ЭВМ.

Объектный подход информационного моделирования предметных областей все более успешно применяется в качестве основы для структуризации их информационных отражений и, в частности, баз знаний.

С++ является языком программирования общего назначения. Именно этот язык хорошо известен своей эффективностью, экономичностью, и переносимостью.

Указанные преимущества **С++** обеспечивают хорошее качество разработки почти любого вида программного продукта.

Использование **С++** в качестве инструментального языка позволяет получать быстрые и компактные программы. Во многих случаях программы, написанные на **С++**, сравнимы по скорости с программами, написанными на языке ассемблера.

Перечислим некоторые существенные особенности языка **C++**:

- **C++** обеспечивает полный набор операторов структурного программирования;
- **C++** предлагает необычно большой набор операций;
- Многие операции **C++** соответствуют машинным командам и поэтому допускают прямую трансляцию в машинный код;
- Разнообразие операций позволяет выбирать их различные наборы для минимизации результирующего кода;
- **C++** поддерживает указатели на переменные и функции;
- Указатель на объект программы соответствует машинному адресу этого объекта;
- Посредством разумного использования указателей можно создавать эффективно выполняемые программы, т.к. указатели позволяют ссылаться на объекты тем же самым путем, как это делает ЭВМ;
- **C++** поддерживает арифметику указателей, и тем самым позволяет осуществлять непосредственный доступ и манипуляции с адресами памяти.

Лексика

Есть шесть классов лексем: идентификаторы, ключевые слова, константы, строки, операторы и прочие разделители. Символы пробела, табуляции и новой строки, а также комментарии (собирательно — «белые места»), как описано ниже, игнорируются, за исключением тех случаев, когда они служат разделителями лексем.

Некое пустое место необходимо для разделения идентификаторов, ключевых слов и констант, которые в противном случае окажутся соприкасающимися.

Если входной поток разобран на лексемы до данного символа, принимается, что следующая лексема содержит наиболее длинную строку символов из тех, что могут составить лексему.

Комментарии

Символы /* задают начало комментария, заканчивающегося символами */. Комментарии не могут быть вложенными.

Символы // начинают комментарий, который заканчивается в конце строки, на которой они появились.

Идентификаторы (имена)

Идентификатор — последовательность букв и цифр произвольной длины; первый символ обязан быть буквой; подчерк '_' считается за букву; буквы в верхнем и нижнем регистрах являются различными.

Ключевые слова

Следующие идентификаторы зарезервированы для использования в качестве ключевых слов и не могут использоваться иным образом:

- asm
- auto
- break
- case
- char
- class
- const
- continue
- default
- delete
- do
- double
- else
- enum
- extern
- float
- for
- friend

- goto
- if
- inline
- int
- long
- new
- operator
- overload
- public
- register
- return
- short
- sizeof
- static
- struct
- switch
- this
- typedef
- union
- unsigned
- virtual
- void
- while

Идентификаторы **signed** и **volatile** зарезервированы для применения в будущем.

Константы

Есть несколько видов констант. Ниже приводится краткая сводка аппаратных характеристик, которые влияют на их размеры.

Целые константы

Целая константа, состоящая из последовательности цифр, считается восьмеричной, если она начинается с 0 (цифры ноль), и десятичной в противном случае. Цифры 8 и 9 не являются восьмеричными цифрами.

Последовательность цифр, которой предшествует **0x** или **0X**, воспринимается как шестнадцатеричное целое.

В шестнадцатеричные цифры входят буквы от **a** или **A** до **f** или **F**, имеющие значения от 10 до 15.

Десятичная константа, значение которой превышает наибольшее машинное целое со знаком, считается длинной (**long**); восьмеричная и шестнадцатеричная константа, значение которой превышает наибольшее машинное целое со знаком, считается **long**; в остальных случаях целые константы считаются **int**.

Явно заданные длинные константы

Десятичная, восьмеричная или шестнадцатеричная константа, за которой непосредственно стоит **l** (латинская буква «эль») или **L**, считается длинной константой.

Символьные константы

Символьная константа состоит из символа, заключенного в одиночные кавычки (апострофы), как, например, **'x'**. Значением символьной константы является численное значение символа в машинном наборе символов (алфавите).

Символьные константы считаются данными типа **int**. Некоторые неграфические символы, одиночная кавычка **'** и обратная косая \, могут быть представлены в соответствие со следующим списком **escape**-последовательностей:

- символ новой строки NL(LF) \n
- горизонтальная табуляция NT \t
- вертикальная табуляция VT \v
- возврат на шаг BS \b
- возврат каретки CR \r
- перевод формата FF \f
- обратная косая \ \\
- одиночная кавычка (апостроф) ' \'

- набор битов 0ddd \ddd
- набор битов 0xddd \xddd

Escape-последовательность **\ddd** состоит из обратной косой, за которой следуют 1, 2 или 3 восьмеричных цифры, задающие значение требуемого символа. Специальным случаем такой конструкции является **\0** (не следует ни одной цифры), задающая пустой символ **NULL**.

Escape-последовательность **\xddd** состоит из обратной косой, за которой следуют 1, 2 или 3 шестнадцатеричных цифры, задающие значение требуемого символа. Если следующий за обратной косой символ не является одним из перечисленных, то обратная косая игнорируется.

Константы с плавающей точкой

Константа с плавающей точкой состоит из целой части, десятичной точки, мантиссы, **e** или **E** и целого показателя степени (возможно, но не обязательно, со знаком). Целая часть и мантисса обе состоят из последовательности цифр.

Целая часть или мантисса (но не обе сразу) может быть опущена; или десятичная точка, или **e (E)** вместе с целым показателем степени (но не обе части одновременно) может быть опущена. Константа с плавающей точкой имеет тип **double**.

Перечислимые константы

Имена, описанные как перечислители, являются константами типа **int**.

Описанные константы

Объект любого типа может быть определен как имеющий постоянное значение во всей области видимости его имени. В случае указателей для достижения этого используется декларатор ***const**; для объектов, не являющихся указателями, используется описатель **const**.

Строки

Строка есть последовательность символов, заключенная в двойные кавычки: «**...**». Строка имеет тип **массив символов** и класс памяти **static**, она инициализируется заданными символами.

Компилятор располагает в конце каждой строки нулевой (пустой) байт **\0** с тем, чтобы сканирующая строку программа могла найти ее конец.

В строке перед символом двойной кавычки " обязательно должен стоять \; кроме того, могут использоваться те же **escape**-последовательности, что были описаны для символьных констант.

И, наконец, символ новой строки может появляться только сразу после \; тогда оба, — \ и символ новой строки, — игнорируются.

Синтаксис

Запись синтаксиса

По синтаксическим правилам записи синтаксические категории выделяются курсивом, а литеральные слова и символы шрифтом постоянной ширины.

Альтернативные категории записываются на разных строках. Необязательный терминальный или нетерминальный символ обозначается нижним индексом «opt», так что **{выражение opt}** указывает на необязательность выражения в фигурных скобках.

Имена и типы

Имя обозначает (денотирует) объект, функцию, тип, значение или метку. Имя вводится в программе описанием. Имя может использоваться только внутри области текста программы, называемой его областью видимости. Имя имеет тип, определяющий его использование.

Объект — это область памяти. Объект имеет класс памяти, определяющий его время жизни. Смысл значения, обнаруженного в объекте, определяется типом имени, использованного для доступа к нему.

Область видимости

Есть четыре вида областей видимости: локальная, файл, программа и класс.

- **Локальная**

Имя, описанное в блоке, локально в этом блоке и может использоваться только в нем после места описания и в охватываемых блоках.

Исключение составляют метки, которые могут использоваться в любом месте функции, в которой они описаны. Имена формальных параметров функции рассматриваются так, как если бы они были описаны в самом внешнем блоке этой функции.

- **Файл**

Имя, описанное вне любого блока или класса, может использоваться в файле, где оно описано, после места описания.

- **Класс**

Имя члена класса локально для его класса и может использоваться только в функции члене этого класса, после примененной к объекту его класса операции, или после примененной к указателю на объект его класса операции **->**.

На статические члены класса и функции члены можно также ссылаться с помощью операции **::** там, где имя их класса находится в области видимости.

Класс, описанный внутри класса, не считается членом, и его имя принадлежит охватывающей области видимости.

Имя может быть скрыто посредством явного описания того же имени в блоке или классе. Имя в блоке или классе может быть скрыто только именем, описанным в охватываемом блоке или классе.

Скрытое нелокальное имя также может использоваться, когда его область видимости указана операцией **::**.

Имя класса, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит **class**, **struct** или **union**. Имя перечисления **enum**, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит **enum**.

Определения

Описание является определением, за исключением тех случаев, когда оно описывает функции, не задавая тела функции, когда оно содержит спецификатор **extern** (1) и в нем нет инициализатора или тела функции, или когда оно является описанием класса.

Компоновка

Имя в файловой области видимости, не описанное явно как **static**, является общим для каждого файла многофайловой программы. Таковым же является имя функции. О таких именах говорится, что они внешние.

Каждое описание внешнего имени в программе относится к тому же объекту, функции, классу, перечислению или значению перечислителя.

Типы, специфицированные во всех описаниях внешнего имени должны быть идентичны. Может быть больше одного определения типа, перечисления, **inline**-функции или несоставного **const**, при условии, что определения идентичны, появляются в разных файлах и все инициализаторы являются константными выражениями.

Во всех остальных случаях должно быть ровно одно определение для внешнего имени в программе.

Реализация может потребовать, чтобы составное **const**, использованное там, где не встречено никакого определения **const**, должно быть явно описано **extern** и иметь в программе ровно одно определение. Это же ограничение может налагаться на **inline**-функции.

Классы памяти

Есть два описываемых класса памяти:

- **автоматический**
- **статический**.

Автоматические объекты локальны для каждого вызова блока и сбрасываются по выходе из него.

Статические объекты существуют и сохраняют свое значение в течение выполнения всей программы.

Некоторые объекты не связаны с именами и их времена жизни явно управляются операторами **new** и **delete**.

Основные типы

Объекты, описанные как символы (**char**), достаточны для хранения любого элемента машинного набора символов, и если принадлежащий этому набору символ хранится в символьной переменной, то ее значение равно целому коду этого символа.

В настоящий момент имеются целые трех размеров, описываемые как **short int**, **int** и **long int**. Более длинные целые (**long int**) предоставляют не меньше памяти, чем более короткие целые (**short int**), но при реализации или длинные, или короткие, или и те и другие могут стать эквивалентными обычным целым.

«Обычные» целые имеют естественный размер, задаваемый архитектурой центральной машины; остальные размеры делаются такими, чтобы они отвечали специальным потребностям.

Каждое перечисление является набором именованных констант. Свойства **enum** идентичны свойствам **int**. Целые без знака, описываемые как **unsigned**, подчиняются правилам арифметики по модулю **2n**, где **n** — число бит в их представлении.

Числа с плавающей точкой одинарной (**float**) и двойной (**double**) точности в некоторых машинных реализациях могут быть синонимами.

Поскольку объекты перечисленных выше типов вполне можно интерпретировать как числа, мы будем говорить о них как об арифметических типах.

Типы **char**, **int** всех размеров и **enum** будут собирательно называться целыми типами. Типы **float** и **double** будут собирательно называться плавающими типами.

Тип данных **void** (пустой) определяет пустое множество значений. Значение (несуществующее) объекта **void** нельзя использовать никаким образом, не могут применяться ни явное, ни неявное преобразования.

Поскольку пустое выражение обозначает несуществующее значение, такое выражение такое выражение может использоваться только как оператор выражение или как левый операнд в выражении с запятой. Выражение может явно преобразовываться к типу **void**.

Производные типы

Кроме основных арифметических типов концептуально существует бесконечно много производных типов, сконструированных из основных типов следующим образом:

- массивы объектов данного типа;
- функции, получающие аргументы данного типа и возвращающие объекты данного типа;
- указатели на объекты данного типа;
- ссылки на объекты данного типа;
- константы, являющиеся значениями данного типа;
- классы, содержащие последовательность объектов различных типов, множество функций для работы с этими объектами и набор ограничений на доступ к этим объектам и функциям;
- структуры, являющиеся классами без ограничений доступа;
- объединения, являющиеся структурами, которые могут в разное время содержать объекты разных типов.

В целом эти способы конструирования объектов могут применяться рекурсивно.

Объект типа **void*** (указатель на **void**) можно использовать для указания на объекты неизвестного типа.

Объекты и LVALUE (адреса)

Объект есть область памяти; **lvalue** (адрес) есть выражение, ссылающееся на объект. Очевидный пример адресного выражения — имя объекта.

Есть операции, дающие адресные выражения: например, если **E** — выражение типа указатель, то ***E** — адресное выражение, ссылающееся на объект, на который указывает **E**.

Термин «**lvalue**» происходит из выражения присваивания **E1=E2**, в котором левый операнд **E1** должен быть адресным (**value**) выражением.

Ниже при обсуждении каждого оператора указывается, требует ли он адресные операнды и возвращает ли он адресное значение.

Символы и целые

Символ или короткое целое могут использоваться, если может использоваться целое. Во всех случаях значение преобразуется к целому.

Преобразование короткого целого к длинному всегда включает в себя знаковое расширение; целые являются величинами со знаком. Содержат символы знаковый разряд или нет, является машинно-зависимым. Более явный тип **unsigned char** ограничивает изменение значения от 0 до машинно-зависимого максимума.

В машинах, где символы рассматриваются как имеющие знак (знаковые), символы множества кода **ASCII** являются положительными.

Однако, символьная константа, заданная восьмеричной **esc**-последовательностью подвергается знаковому расширению и может стать отрицательным числом; так например, **'\377'** имеет значение **-1**.

Когда длинное целое преобразуется в короткое или в **char**, оно урезается влево; избыточные биты просто теряются.

Float и double

Для выражений **float** могут выполняться действия арифметики с плавающей точкой одинарной точности. Преобразования между числами одинарной и двойной точности выполняются настолько математически корректно, насколько позволяет аппаратура.

Плавающие и целые

Преобразования плавающих значений в интегральный тип имеет склонность быть машинно-зависимым. В частности, направление усечения отрицательных чисел различается от машины к машине. Если предоставляемого пространства для значения не хватает, то результат не определен.

Преобразование интегрального значения в плавающий тип выполняются хорошо. При нехватке в аппаратной реализации требуемых бит, возникает некоторая потеря точности.

Указатели и целые

Выражение целого типа можно прибавить к указателю или вычесть из него; в таком случае первый преобразуется, как указывается при обсуждении операции сложения.

Можно производить вычитание над двумя указателями на объекты одного типа; в этом случае результат преобразуется к типу **int** или **long** в зависимости от машины.

Unsigned

Всегда при сочетании целого без знака и обычного целого обычное целое преобразуется к типу **unsigned** и результат имеет тип **unsigned**.

Значением является наименьшее целое без знака, равное целому со знаком (**mod 2****(размер слова)) (т.е. по модулю **2****(размер слова)). В дополнительном двоичном представлении это преобразование является пустым, и никаких реальных изменений в двоичном представлении не происходит.

При преобразовании целого без знака в длинное значение результата численно совпадает со значением целого без знака. Таким образом, преобразование сводится к дополнению нулями слева.

Преобразования

Арифметические преобразования

Большое количество операций вызывают преобразования и дают тип результата одинаковым образом. Этот стереотип будет называться **«обычным арифметическим преобразованием»**.

Во-первых, любые операнды типа **char**, **unsigned char** или **short** преобразуются к типу **int**.

Далее, если один из операндов имеет тип **double**, то другой преобразуется к типу **double** и тот же тип имеет результат.

Иначе, если один из операндов имеет тип **unsigned long**, то другой преобразуется к типу **unsigned long** и таков же тип результата.

Иначе, если один из операндов имеет тип **long**, то другой преобразуется к типу **long** и таков же тип результата.

Иначе, если один из операндов имеет тип **unsigned**, то другой преобразуется к типу **unsigned** и таков же тип результата.

Иначе оба операнда должны иметь тип **int** и таков же тип результата.

Преобразования указателей

Везде, где указатели присваиваются, инициализируются, сравниваются и т.д. могут выполняться следующие преобразования.

- Константа 0 может преобразовываться в указатель, и гарантируется, что это значение породит указатель, отличный от указателя на любой объект.
- Указатель любого типа может преобразовываться в **void***.
- Указатель на класс может преобразовываться в указатель на открытый базовый класс этого класса.
- Имя вектора может преобразовываться в указатель на его первый элемент.
- Идентификатор, описанный как «**функция, возвращающая ...**», всегда, когда он не используется в позиции имени функции в вызове, преобразуется в «**указатель на функцию, возвращающую ...**».

Преобразования ссылок

Везде, где инициализируются ссылки, может выполняться следующее преобразование.

Ссылка на класс может преобразовываться в ссылку на открытый базовый класс этого класса.

Выражения и операции

Приоритет операций в выражениях такой же, как и порядок главных подразделов в этом разделе, наибольший приоритет у первого. Внутри каждого подраздела операции имеют одинаковый приоритет.

В каждом подразделе для рассматриваемых в нем операций определяется их левая или правая ассоциативность (порядок обработки операндов). Приоритет и ассоциативность всех операций собран вместе в описании грамматики. В остальных случаях порядок вычисления выражения не определен. Точнее, компилятор волен вычислять подвыражения в том порядке, который он считает более эффективным, даже если подвыражения вызывают побочные эффекты.

Порядок возникновения побочных эффектов не определен. Выражения, включающие в себя коммутативные и ассоциативные операции (*****, **+**, **&**, **|**, **^**), могут быть реорганизованы произвольным образом, даже при наличии скобок; для задания определенного порядка вычисления выражения необходимо использовать явную временную переменную.

Обработка переполнения и контроль деления при вычислении выражения машинно-зависимы. В большинстве существующих реализаций **С++** переполнение целого игнорируется; обработка деления на 0 и всех исключительных ситуаций с числами с плавающей точкой различаются от машины к машине и обычно могут регулироваться библиотечными функциями.

Кроме стандартного значения, операции могут быть перегружены, то есть, могут быть заданы их значения для случая их применения к типам, определяемым пользователем.

Основные выражения

Идентификатор есть первичное выражение, причем соответственно описанное. **Имя_функции_операции** есть идентификатор со специальным значением.

Операция **::**, за которой следует идентификатор из файловой области видимости, есть то же, что и идентификатор.

Это позволяет ссылаться на объект даже в том случае, когда его идентификатор скрыт.

Typedef-имя, за которым следует **::**, после чего следует идентификатор, является первичным выражением. **Typedef**-имя должно обозначать класс, и идентификатор должен обозначать член этого класса. Его тип специфицируется описанием идентификатора.

Typedef-имя может быть скрыто именем, которое не является именем типа. В этом случае **typedef**-имя все равно может быть найдено и его можно использовать.

Константа является первичным выражением. Ее тип должен быть **int**, **long** или **double** в зависимости от ее формы.

Строка является первичным выражением. Ее тип — «**массив символов**». Обычно он сразу же преобразуется в указатель на ее первый символ.

Ключевое слово **this** является локальной переменной в теле функции члена. Оно является указателем на объект, для которого функция была вызвана.

Выражение, заключенное в круглые скобки, является первичным выражением, чей тип и значение те же, что и у не заключенного в скобки выражения. Наличие скобок не влияет на то, является выражение **lvalue** или нет.

Первичное выражение, за которым следует выражение в квадратных скобках, является первичным выражением. Интуитивный смысл — индекс. Обычно первичное выражение имеет тип «**указатель на ...**», индексирующее выражение имеет тип **int** и тип результата есть «**...**».

Выражение **E1[E2]** идентично (по определению) выражению ***((E1)+(E2))**.

Вызов функции является первичным выражением, за которым следуют скобки, содержащие список (возможно, пустой) разделенных запятыми выражений, составляющих фактические параметры для функции. Первичное выражение должно иметь тип «**функция, возвращающая ...**» или «**указатель на функцию, возвращающую ...**», и результат вызова функции имеет тип «**...**».

Каждый формальный параметр инициализируется фактическим параметром. Выполняются стандартные и определяемые пользователем преобразования. Функция может изменять значения своих формальных параметров, но эти

изменения не могут повлиять на значения фактических параметров за исключением случая, когда формальный параметр имеет ссылочный тип.

Функция может быть описана как получающая меньше или больше параметров, чем специфицировано в описании функции. Каждый фактический параметр типа **float**, для которого нет формального параметра, преобразуются к типу **double**; и, как обычно, имена массивов преобразуются к указателям. Порядок вычисления параметров не определен языком; имейте в виду различия между компиляторами. Допустимы рекурсивные вызовы любых функций.

Первичное выражение, после которого стоит точка, за которой следует идентификатор (или идентификатор, уточненный **typedef**-именем с помощью операции **::**) является выражением. Первое выражение должно быть объектом класса, а идентификатор должен именовать член этого класса.

Значением является именованный член объекта, и оно является адресным, если первое выражение является адресным.

Следует отметить, что **«классовые объекты»** могут быть **структурами** или **объединениями**.

Первичное выражение, после которого стоит стрелка (**->**), за которой следует идентификатор (или идентификатор, уточненный **typedef**-именем с помощью операции **::**) является выражением.

Первое выражение должно быть указателем на объект класса, а идентификатор должен именовать член этого класса. Значение является адресом, ссылающимся на именованный член класса, на который указывает указательное выражение.

Так, выражение **E1->MOS** есть то же, что и **(*E1).MOS**. Если первичное выражение дает значение типа **«указатель на ...»**, значением выражения был объект, обозначаемый ссылкой. Ссылку можно считать именем объекта.

Унарные операции

Унарная операция * означает косвенное обращение: выражение должно быть указателем и результатом будет **lvalue**, ссылающееся на объект, на который указывает выражение. Если

выражение имеет тип **«указатель на ...»**, то тип результата есть «**...**».

Результатом унарной операции **&** является указатель на объект, на который ссылается операнд. Операнд должен быть **lvalue**. Если выражение имеет тип «**...**», то тип результата есть **«указатель на ...»**.

Результатом унарной операции + является значение ее операнда после выполнения обычных арифметических преобразований. Операнд должен быть арифметического типа.

Результатом унарной операции является отрицательное значение ее операнда. Операнд должен иметь целый тип. Выполняются обычные арифметические преобразования. Отрицательное значение беззнаковой величины вычислятся посредством вычитания ее значения из **2n**, где **n** — число битов в целом типа **int**.

Результатом операции логического отрицания **!** является 1, если значение операнда 0, и 0, если значение операнда не 0. Результат имеет тип **int**. Применима к любому арифметическому типу или к указателям.

Операция ~ дает дополнение значения операнда до единицы. Выполняются обычные арифметические преобразования. Операнд должен иметь интегральный тип.

Увеличение и Уменьшение

Операнд префиксного ++ получает приращение. Операнд должен быть адресным. Значением является новое значение операнда, но оно не адресное. Выражение **++x** эквивалентно **x+=1**. По поводу данных о преобразованиях смотрите обсуждение операций сложения и присваивания.

Операнд префиксного **--** уменьшается аналогично действию префиксной операции ++.

Значение, получаемое при использовании постфиксного ++, есть значение операнда. Операнд должен быть адресным.

После того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

Значение, получаемое при использовании постфиксной **--**, есть значение операнда. Операнд должен быть адресным. После

того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

Sizeof

Операция **sizeof** дает размер операнда в байтах. (Байт не определяется языком иначе, чем через значение **sizeof**.

Однако, во всех существующих реализациях байт есть пространство, необходимое для хранения **char**).

При применении к массиву результатом является полное количество байтов в массиве. Размер определяется из описаний объектов, входящих в выражение. Семантически это выражение является беззнаковой константой и может быть использовано в любом месте, где требуется константа.

Операцию **sizeof** можно также применять к заключенному в скобки имени типа. В этом случае она дает размер, в байтах, объекта указанного типа.

Явное Преобразование Типа

Простое_имя_типа, возможно, заключенное в скобки, за которым идет заключенное в скобки выражение (или **список_выражений**, если тип является классом с соответствующим образом описанным конструктором) влечет преобразование значения выражения в названный тип.

Чтобы записать преобразование в тип, не имеющий простого имени, **имя_типа** должно быть заключено в скобки. Если имя типа заключено в скобки, выражение заключать в скобки необязательно. Такая запись называется приведением к типу.

Указатель может быть явно преобразован к любому из интегральных типов, достаточно по величине для его хранения. То, какой из **int** и **long** требуется, является машинно-зависимым. Отображающая функция также является машинно-зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине.

Объект интегрального типа может быть явно преобразован в указатель. Отображающая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машиннозависимой.

Указатель на один тип может быть явно преобразован в указатель на другой тип. Использование полученного в результате указателя может привести к исключительной ситуации адресации, если исходный указатель не указывает на объект, соответствующим образом выровненный в памяти.

Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений. Различные машины могут различаться по числу бит в указателях и требованиям к выравниванию объектов.

Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

Объект может преобразовываться в объект класса только если был описан соответствующий конструктор или операция преобразования.

Объект может явно преобразовываться в ссылочный тип **&X**, если указатель на этот объект может явно преобразовываться в **X***.

Свободная Память

Операция **new** создает объект типа **имя_типа**, к которому он применен. Время жизни объекта, созданного с помощью **new**, не ограничено областью видимости, в которой он создан. Операция **new** возвращает указатель на созданный ей объект.

Когда объект является массивом, возвращается указатель на его первый элемент. Например, и **new int** и **new int[10]** возвращают **int***.

Для объектов некоторых классов надо предоставлять инициализатор. Операция **new** для получения памяти вызывает функцию:

```
void* operator new (long);
```

Параметр задает требуемое число байтов. Память будет инициализирована. Если **operator new()** не может найти требуемое количество памяти, то она возвращает ноль.

Операция **delete** уничтожает объект, созданный операцией **new**. Ее результат является **void**. Операнд **delete** должен быть указателем, возвращенным **new**. Результат применения **delete** к указателю, который не был получен с помощью операции **new**.

Однако уничтожение с помощью **delete** указателя со значением ноль безвредно.

Чтобы освободить указанную память, операция **delete** вызывает функцию

```
void operator delete (void*);
```

В форме

```
delete [ выражение ] выражение
```

второй параметр указывает на вектор, а первое выражение задает число элементов этого вектора. Задание числа элементов является избыточным за исключением случаев уничтожения векторов некоторых классов.

Мультипликативные операции

Мультипликативные операции *, / и % группируют слева направо. Выполняются обычные арифметические преобразования.

Синтаксис:

```
выражение * выражение
выражение / выражение
выражение  %  выражение
```

Бинарная операция * определяет умножение. Операция * ассоциативна и выражения с несколькими умножениями на одном уровне могут быть реорганизованы компилятором.

Бинарная операция / определяет деление. При делении положительных целых округление осуществляется в сторону 0, но если какой-либо из операндов отрицателен, то форма округления является машинно-зависимой.

На всех машинах, охватываемых данным руководством, остаток имеет тот же знак, что и делимое. Всегда истинно, что **(a/b)*b + a%b** равно **a** (если **b** не 0).

Бинарная операция % дает остаток от деления первого выражения на второе. Выполняются обычные арифметические преобразования. Операнды не должны быть числами с плавающей точкой.

Аддитивные операции

Аддитивные операции + и — группируют слева направо. Выполняются обычные арифметические преобразования.

Каждая операция имеет некоторые дополнительные возможности, связанные с типами.

Синтаксис:

```
выражение + выражение
выражение  -  выражение
```

Результатом операции **+** является сумма операндов. Можно суммировать указатель на объект массива и значение целого типа. Последнее во всех случаях преобразуется к смещению адреса с помощью умножения его на длину объекта, на который указывает указатель.

Результатом является указатель того же типа, что и исходный указатель, указывающий на другой объект того же массива и соответствующим образом смещенный от первоначального объекта. Так, если **P** есть указатель на объект массива, то выражение **P+1** есть указатель на следующий объект массива.

Никакие другие комбинации типов для указателей не допустимы.

Операция **+** ассоциативна и выражение с несколькими умножениями на одном уровне может быть реорганизовано компилятором.

Результатом операции — является разность операндов. Выполняются обычные арифметические преобразования. Кроме того, значение любого целого типа может вычитаться из указателя, в этом случае применяются те же преобразования, что и к сложению.

Если вычитаются указатели на объекты одного типа, то результат преобразуется (посредством деления на длину объекта) к целому, представляющему собой число объектов, разделяющих объекты, указанные указателями.

В зависимости от машины результирующее целое может быть или типа **int**, или типа **long**.

Вообще говоря, это преобразование будет давать неопределенный результат кроме тех случаев, когда указатели указывают на объекты одного массива, поскольку указатели, даже на объекты одинакового типа, не обязательно различаются на величину, кратную длине объекта.

Операции сдвига

Операции сдвига << и >> группируют слева направо. Обе выполняют одно обычное арифметическое преобразование над своими операндами, каждый из которых должен быть целым. В этом случае правый операнд преобразуется к типу **int**; тип результата совпадает с типом левого операнда. Результат не определен, если правый операнд отрицателен или больше или равен длине объекта в битах.

Синтаксис:

```
выражение << выражение
выражение >> выражение
```

Значением **E1 << E2** является **E1** (рассматриваемое как битовое представление), сдвинутое влево на **E2** битов; освободившиеся биты заполняются нулями. Значением **E1 >> E2** является **E1**, сдвинутое вправо на **E2** битовых позиций.

Гарантируется, что сдвиг вправо является **логическим** (заполнение нулями), если **E1** является **unsigned**; в противном случае он может быть **арифметическим** (заполнение копией знакового бита).

Операции отношения

Операции отношения (сравнения) группируют слева направо, но этот факт не очень-то полезен: **a < b < c** не означает то, чем кажется.

Синтаксис:

```
выражение < выражение
выражение > выражение
выражение <= выражение
выражение >= выражение
```

Операции **<** (меньше чем), **>** (больше чем), **<=** и **>=** все дают **0**, если заданное соотношение ложно, и **1**, если оно истинно. Тип результата **int**.

Выполняются обычные арифметические преобразования. Могут сравниваться два указателя; результат зависит от относительного положения объектов, на которые указывают указатели, в адресном пространстве. Сравнение указателей переносимо только если указатели указывают на объекты одного массива.

Операции равенства

Синтаксис:

```
выражение == выражение
выражение  != выражение
```

Операции **==** и **!=** в точности аналогичны операциям сравнения за исключением их низкого приоритета. (Так, a **< b == c < d** есть **1** всегда, когда **a < b** и **c < d** имеют одинаковое истинностное значение).

Указатель может сравниваться с **0**.

Операция побитовое И

Синтаксис:

```
выражение & выражение
```

Операция **&** ассоциативна, и выражения, содержащие **&**, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция **И** операндов. Операция применяется только к целым операндам.

Операция побитовое исключающее ИЛИ

Синтаксис:

```
выражение ^ выражение
```

Операция ^ ассоциативна, и выражения, содержащие ^, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция исключающее **ИЛИ** операндов. Операция применяется только к целым операндам.

Операция побитовое включающее ИЛИ

Синтаксис:

```
выражение | выражение
```

Операция **|** ассоциативна, и выражения, содержащие **|**, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция включающее **ИЛИ** операндов. Операция применяется только к целым операндам.

Операция логическое И

Синтаксис:

```
выражение && выражение
```

Операция **&&** группирует слева направо. Она возвращает **1**, если оба операнда ненулевые, и **0** в противном случае. В противоположность операции **&** операция **&&** гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд есть **0**.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип **int**.

Операция логическое ИЛИ

Синтаксис:

```
выражение || выражение
```

Операция **||** группирует слева направо. Она возвращает **1**, если хотя бы один из ее операндов ненулевой, и **0** в противном случае. В противоположность операции **|** операция **||** гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд не есть **0**.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип **int**.

Условная операция

Синтаксис:

```
выражение ? выражение : выражение
```

Условная операция группирует слева направо. Вычисляется первое выражение, и если оно не **0**, то результатом является значение второго выражения, в противном случае значение третьего выражения.

Если это возможно, то выполняются обычные арифметические преобразования для приведения второго и третьего выражения к общему типу.

Если это возможно, то выполняются преобразования указателей для приведения второго и третьего выражения к общему типу. Вычисляется только одно из второго и третьего выражений.

Операции присваивания

Есть много операций присваивания, все группируют слева направо. Все в качестве левого операнда требуют **lvalue**, и тип выражения присваивания тот же, что и у его левого операнда. Это **lvalue** не может ссылаться на константу (имя массива, имя функции или **const**).

Значением является значение, хранящееся в левом операнде после выполнения присваивания.

Синтаксис:

```
выражение операция_присваивания выражение
```

Где **операция_присваивания** — одна из:

- =
- +=
- -=
- *=
- /=
- %=
- >>=
- <<=
- &=
- ~=
- |=

В простом присваивании с = значение выражения замещает собой значение объекта, на который ссылается операнд в левой части. Если оба операнда имеют арифметический тип, то при подготовке к присваиванию правый операнд преобразуется к типу левого.

Если аргумент в левой части имеет указательный тип, аргумент в правой части должен быть того же типа или типа, который может быть преобразован к нему.

Оба операнда могут быть объектами одного класса. Могут присваиваться объекты некоторых производных классов.

Присваивание объекту типа «**указатель на ...**» выполнит присваивание объекту, денотируемому ссылкой.

Выполнение выражения вида **E1 op= E2** можно представить себе как эквивалентное **E1 = E1 op (E2)**; но **E1** вычисляется только один раз.

В **+=** и **-=** левый операнд может быть указателем, и в этом случае (интегральный) правый операнд преобразуется так, как объяснялось выше; все правые операнды и не являющиеся указателями левые должны иметь арифметический тип.

Операция запятая

Синтаксис:

```
выражение , выражение
```

Пара выражений, разделенных запятой, вычисляется слева направо, значение левого выражения теряется. Тип и значение результата являются типом и значением правого операнда. Эта операция группирует слева направо.

В контексте, где запятая имеет специальное значение, как например в списке фактических параметров функции и в списке инициализаторов, операция запятая, как она описана в этом разделе, может появляться только в скобках.

Например:

```
f (a,(t=3,t+2),c)
```

имеет три параметра, вторым из которых является значение 5.

Перегруженные операции

Большинство операций может быть перегружено, то есть, описано так, чтобы они получали в качестве операндов объекты классов. Изменить приоритет операций невозможно. Невозможно изменить смысл операций при применении их к неклассовым объектам.

Предопределенный смысл операций **=** и **&** (унарной) при применении их к объектам классов может быть изменен.

Эквивалентность операций, применяемых к основным типам (например, **++a** эквивалентно **a+=1**), не обязательно выполняется для операций, применяемых к классовым типам. Некоторые операции, например, присваивание, в случае применения к основным типам требуют, чтобы операнд был **lvalue**; это не требуется для операций, описанных для классовых типов.

Унарные операции

Унарная операция, префиксная или постфиксная, может быть определена или с помощью функции члена, не получающей параметров, или с помощью функции друга, получающей один параметр, но не двумя способами одновременно.

Так, для любой унарной операции @, **x@** и **@x** могут интерпретироваться как **x.операция@()** или **операция@(x)**. При перегрузке операций ++ и -- невозможно различить префиксное и постфиксное использование.

Бинарные операции

Бинарная операция может быть определена или с помощью функции члена, получающей один параметр, или с помощью функции друга, получающей два параметра, но не двумя способами одновременно. Так, для любой бинарной операции @, **x@y** может быть проинтерпретировано как **x.операция@(y)** или **операция@(x,y)**.

Особые операции

Вызов функции:

```
первичное_выражение (список_выражений opt)
```

и индексирование

```
первичное_выражение [выражение]
```

считаются бинарными операциями. Именами определяющей функции являются соответственно **operator()** и **operator[]**. Обращение **x(arg)** интерпретируется как **x.operator()(arg)** для классового объекта **x**. Индексирование **x[y]** интерпретируется как **x.operator[](y)**.

Описания

Описания используются для определения интерпретации, даваемой каждому идентификатору; они не обязательно резервируют память, связанную с идентификатором. Описания имеют вид:

```
спецификаторы_описания opt список_описателей opt;
описание_имени
asm_описание
```

Описатели в **списке_описателей** содержат идентификаторы, подлежащие описанию. **Спецификаторы_описания** могут быть опущены только в определениях внешних функций или в описаниях внешних функций. Список описателей может быть пустым только при описании класса или перечисления, то есть, когда **спецификаторы_описания —** это **class_спецификатор** или **enum_спецификатор**.

Список должен быть внутренне непротиворечив в описываемом ниже смысле.

Спецификаторы класса памяти

Спецификаторы **«класса памяти»** (**sc**-спецификатор) это:

- auto
- static
- extern
- register

Описания, использующие спецификаторы **auto**, **static** и **register** также служат определениями тем, что они вызывают резервирование соответствующего объема памяти. Если описание extern не является определением, то где-то еще должно быть определение для данных идентификаторов.

Описание **register** лучше всего представить как описание **auto** (автоматический) с подсказкой компилятору, что описанные переменные усиленно используются. Подсказка может быть проигнорирована. К ним не может применяться операция получения адреса **&**.

Спецификаторы **auto** или **register** могут применяться только к именам, описанным в блоке, или к формальным параметрам. Внутри блока не может быть описаний ни статических функций, ни статических формальных параметров.

В описании может быть задан максимум один **sc_спецификатор**. Если в описании отсутствует **sc_спецификатор**, то класс памяти принимается автоматическим внутри функции и статическим вне. Исключение: функции не могут быть автоматическими.

Спецификаторы **static** и **extern** могут использоваться только для имен объектов и функций.

Некоторые спецификаторы могут использоваться только в описаниях функций:

- overload
- inline
- virtual

Спецификатор перегрузки **overload** делает возможным использование одного имени для обозначения нескольких функций.

Спецификатор **inline** является только подсказкой компилятору, не влияет на смысл программы и может быть проигнорирован.

Он используется, чтобы указать на то, что при вызове функции **inline —** подстановка тела функции предпочтительнее обычной реализации вызова функции. Функция, определенная внутри описания класса, является **inline** по умолчанию. Спецификатор **virtual** может использоваться только в описаниях членов класса.

Спецификатор **friend** используется для отмены правил сокрытия имени для членов класса и может использоваться только внутри описаний классов. С помощью спецификатора **typedef** вводится имя для типа.

Спецификаторы Типа

Спецификаторами типов (**спецификатор_типа**) являются:

```
спецификатор_типа:
простое_имя_типа
class_спецификатор
enum-спецификатор
сложный_спецификатор_типа
const
```

Слово **const** можно добавлять к любому допустимому **спецификатору_типа**. В остальных случаях в описании может быть дано не более одного **спецификатора_типа**. Объект типа **const** не является **lvalue**. Если в описании опущен спецификатор типа, он принимается **int**.

Простое_имя_типа — это:

- char
- short
- int
- long
- unsigned
- float
- double
- const
- void

Слова **long**, **short** и **unsigned** можно рассматривать как прилагательные. Они могут применяться к типу **int**; **unsigned** может также применяться к типам **char**, **short** и **long**.

Сложный_спецификатор_типа — это:

- ключ typedef-имя
- ключ идентификатор

Ключ:

- class
- struct
- union
- enum

Сложный спецификатор типа можно использовать для ссылки на имя класса или перечисления там, где имя может быть скрыто локальным именем.

Например:

```
class x { ... };
void f(int x)
{
   class x a;
// ...
```

Если имя класса или перечисления ранее описано не было, **сложный_спецификатор_типа** работает как **описание_имени**.

Описатели

Список_описателей, появляющийся в описании, есть разделенная запятыми последовательность описателей, каждый из которых может иметь инициализатор.

```
список_описателей:
иниц_описатель
иниц_описатель , список_описателей
```

Где **иниц_описатель**:

```
описатель инициализатор opt
```

Спецификатор в описании указывает тип и класс памяти объектов, к которым относятся описатели.

Описатели имеют синтаксис:

```
описатель:
оп_имя
( описатель )
* const opt описатель
& const opt описатель
описатель
( список_описаний_параметров )
описатель
[ константное_выражение opt ]
```

Где **оп-имя**:

```
простое_оп_имя
typedef-имя  ::  простое_оп_имя
```

А **простое_оп_имя** — это:

```
идентификатор
typedef-имя
~ typedef-имя
имя_функции_операции
имя_функции_преобразования
```

Группировка та же, что и в выражениях.

Смысл описателей

Каждый описатель считается утверждением того, что если в выражении возникает конструкция, имеющая ту же форму, что и описатель, то она дает объект указанного типа и класса памяти. Каждый описатель содержит ровно одно **оп_имя**; оно определяет описываемый идентификатор. За исключением описаний

некоторых специальных функций, **оп_имя** будет простым идентификатором.

Если в качестве описателя возникает ничем не снабженный идентификатор, то он имеет тип, указанный спецификатором, возглавляющим описание. Описатель в скобках эквивалентен описателю без скобок, но связку сложных описателей скобки могут изменять.

Теперь представим себе описание:

```
T D1
```

где **T** — спецификатор типа (как **int** и т.д.), а **D1 —** описатель. Допустим, что это описание заставляет идентификатор иметь тип **«... T»**, где **«...»** пусто, если идентификатор **D1** есть просто обычный идентификатор (так что тип **x** в **«int x»** есть просто **int**). Тогда, если **D1** имеет вид

```
*D
```

то тип содержащегося идентификатора есть «**... указатель на T**»

Если **D1** имеет вид

```
* const D
```

то тип содержащегося идентификатора есть **«... константный указатель на T»**, то есть, того же типа, что и ***D**, но не **lvalue**.

Если **D1** имеет вид

```
&D
```

или

```
& const D
```

то тип содержащегося идентификатора есть **«... ссылка на T»**. Поскольку ссылка по определению не может быть **lvalue**, использование **const** излишне. Невозможно иметь ссылку на **void (void&)**.

Если **D1** имеет вид

```
D (список_описаний_параметров)
```

то содержащийся идентификатор имеет тип **«... функция, принимающая параметр типа список_описаний_параметров и возвращающая T»**.

```
список_описаний_параметров:
список_описаний_парам opt ... opt
список_описаний_парам:
список_описаний_парам , описание_параметра
```

```
описание_параметра
описание_параметра:
спецификаторы_описания описатель
спецификаторы_описания
описатель = выражение
спецификаторы_описания абстракт_описатель
спецификаторы_описания абстракт_описатель = выражение
```

Если **список_описаний_параметров** заканчивается многоточием, то о числе параметров известно лишь, что оно равно или больше числа специфицированных типов параметров; если он пуст, то функция не получает ни одного параметра.

Все описания для функции должны согласовываться и в типе возвращаемого значения, а также в числе и типе параметров.

Список_описаний_параметров используется для проверки и преобразования фактических параметров и для контроля присваивания указателю на функцию. Если в описании параметра специфицировано выражение, то это выражение используется как параметр по умолчанию.

Параметры по умолчанию будут использоваться в вызовах, где опущены стоящие в хвосте параметры. Параметр по умолчанию не может переопределяться более поздними описаниями. Однако, описание может добавлять параметры по умолчанию, не заданные в предыдущих описаниях.

Идентификатор может по желанию быть задан как имя параметра. Если он присутствует в описании функции, его использовать нельзя, поскольку он сразу выходит из области видимости. Если он присутствует в определении функции, то он именует формальный параметр.

Если **D1** имеет вид:

```
D[константное_выражение]
```

или

```
D[]
```

то тип содержащегося идентификатора есть «**... массив объектов типа T**». В первом случае **константное_выражение** есть выражение, значение которого может быть определено во время компиляции, и тип которого **int**.

Если подряд идут несколько спецификаций «**массив из**», то создается многомерный массив; константное выражение,

определяющее границы массива, может быть опущено только для первого члена последовательности.

Этот пропуск полезен, когда массив является внешним, и настоящее определение, которое резервирует память, находится в другом месте.

Первое константное выражение может также быть опущено, когда за описателем следует инициализация. В этом случае используется размер, вычисленный исходя из числа начальных элементов.

Массив может быть построен из одного из основных типов, из указателей, из структуры или объединения или из другого массива (для получения многомерного массива).

Не все возможности, которые позволяет приведенный выше синтаксис, допустимы. Ограничения следующие: функция не может возвращать массив или функцию, хотя она может возвращать указатели на эти объекты; не существует массивов функций, хотя могут быть массивы указателей на функции.

Примеры

Описание:

```
int i;
int *ip;
int f ();
int *fip ();
int (*pfi) ();
```

описывает целое **i**, указатель **ip** на целое, функцию **f**, возвращающую целое, функцию **fip**, возвращающую указатель на целое, и указатель **pfi** на функцию, возвращающую целое. Особенно полезно сравнить последние две. Цепочка ***fip()** есть ***(fip())**, как предполагается в описании, и та же конструкция требуется в выражении, вызов функции **fip**, и затем косвенное использование результата через (указатель) для получения целого.

В описателе **(*pfi)()** внешние скобки необходимы, поскольку они также входят в выражение, для указания того, что функция получается косвенно через указатель на функцию, которая затем вызывается; это возвращает целое.

Функции **f** и **fip** описаны как не получающие параметров, и **fip** как указывающая на функцию, не получающую параметров.

Описание:

```
const a = 10, *pc = &a, *const
cpc = pc;
int b, *const cp = &b;
```

описывает

- **a**: целую константу
- **pc**: указатель на целую константу
- **cpc**: константный указатель на целую константу
- **b**: целое
- **cp**: константный указатель на целое.

Значения **a**, **cpc** и **cp** не могут быть изменены после инициализации. Значение **pc** может быть изменено, как и объект, указываемый **cp**.

Примеры недопустимых выражений:

```
a = 1;
a++;
*pc = 2;
cp = &a;
cpc++;
```

Примеры допустимых выражений:

```
b = a;
*cp = a;
pc++;
pc = cpc;
```

Описание:

```
fseek (FILE*,long,int);
```

описывает функцию, получающую три параметра специальных типов. Поскольку тип возвращаемого значения не определен, принимается, что он **int**.

Описание:

```
point (int = 0,int = 0);
```

описывает функцию, которая может быть вызвана без параметров, с одним или двумя параметрами типа **int**.

Например:

```
point (1,2);
point (1)
/* имеет смысл point (1,0); */
point ()
/* имеет смысл point (0,0); */
```

Описание:

```
printf (char* ... );
```

описывает функцию, которая может быть вызываться с различными числом и типами параметров.

Например:

```
printf ("hello, world");
printf ("a=%d b=%d",a,b);
printf ("string=%s",st);
```

Однако, она всегда должна иметь своим первым параметром **char***.

В качестве другого примера,

```
float fa[17], *afp[17];
```

описывает массив чисел с плавающей точкой и массив указателей на числа с плавающей точкой.

И, наконец,

```
static int x3d[3][5][7];
```

описывает массив целых, размером 3x6x7.

Совсем подробно:

- **x3d** является массивом из трех элементов;
- каждый из элементов является массивом из пяти элементов;
- каждый из последних элементов является массивом из семи целых.

Появление каждого из выражений **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** может быть приемлемо.

Первые три имеют тип **«массив»**, последний имеет тип **int**.

Описания классов

Класс специфицирует тип. Его имя становится **typedef**-имя, которое может быть использовано даже внутри самого спецификатора класса. Объекты класса состоят из последовательности членов.

Синтаксис:

```
заголовок_класса
{ список_членов opt }
заголовок_класса
{ список_членов opt public  :
список_членов opt }
```

Где **заголовок_класса**:

```
агрег идентификатор opt
```

Где **идентификатор opt**:

```
public opt typedef-имя
```

А **агрег** может иметь вид:

- class
- struct
- union

Структура является классом, все члены которого общие. Объединение является классом, содержащим в каждый момент только один член. Список членов может описывать члены вида: данные, функция, класс, определение типа, перечисление и поле.

Список членов может также содержать описания, регулирующие видимость имен членов.

Синтаксис:

```
описание_члена
список_членов opt
Описание_члена:
спецификаторы_описания opt описатель_члена;
Описатель_члена:
описатель
идентификатор opt :
константное_выражение
```

Члены, являющиеся классовыми объектами, должны быть объектами предварительно полностью описанных классов. В

частности, класс **cl** не может содержать объект класса **cl**, но он может содержать указатель на объект класса **cl**. Имена объектов в различных классах не конфликтуют между собой и с обычными переменными.

Вот простой пример описания структуры:

```
struct tnode
   {
          char tword[20];
          int count;
          tnode *left;
          tnode *right;
   };
```

содержащей массив из 20 символов, целое и два указателя на такие же структуры.

Если было дано такое описание, то описание

```
tnode s, *sp
```

описывает **s** как структуру данного сорта и **sp** как указатель на структуру данного сорта.

При наличии этих описаний выражение

```
sp->count
```

ссылается на поле **count** структуры, на которую указывает **sp**;

```
s.left
```

ссылается на указатель левого поддерева структуры **s**;

```
s.right->tword[0]
```

ссылается на первый символ члена **tword** правого поддерева структуры **s**.

Статические члены

Член-данные класса может быть **static**; члены-функции не могут. Члены не могут быть **auto**, **register** или **extern**. Есть единственная копия статического члена, совместно используемая всеми членами класса в программе. На статический член **mem** класса **cl** можно ссылаться **cl:mem**, то есть без ссылки на объект. Он существует, даже если не было создано ни одного объекта класса **cl**.

Функции-члены

Функция, описанная как член, (без спецификатора **friend**) называется функцией членом и вызывается с помощью синтаксиса члена класса.

Например:

```
struct tnode
{
char tword[20];
int count;
tnode *left;
tnode *right;
void set (char* w,tnode* l,tnode* r);
};
tnode n1, n2;
n1.set ("asdf",&n2,0);
n2.set ("ghjk",0,0);
```

Определение функции члена рассматривается как находящееся в области видимости ее класса. Это значит, что она может непосредственно использовать имена ее класса. Если определение функции члена находится вне описания класса, то имя функции члена должно быть уточнено именем класса с помощью записи

```
typedef-имя . простое_оп_имя
```

Например:

```
void tnode.set (char* w,tnode* l,tnode* r)
{
count = strlen (w);
if (sizeof (tword) <= count) error ("tnode string too
long");
strcpy (tword,w);
left = l;
right = r;
}
```

Имя функции **tnode.set** определяет то, что множество функций является членом класса **tnode**. Это позволяет использовать имена членов **word**, **count**, **left** и **right**.

В функции члене имя члена ссылается на объект, для которого была вызвана функция. Так, в вызове **n1.set(...) tword** ссылается на **n1.tword**, а в вызове **n2.set(...)** он ссылается на

n2.tword. В этом примере предполагается, что функции **strlen**, **error** и **strcpy** описаны где-то в другом месте как внешние функции.

В члене функции ключевое слово **this** указывает на объект, для которого вызвана функция. Типом **this** в функции, которая является членом класса **cl**, является **cl***.

Если **mem** — член класса **cl**, то **mem** и **this->mem** — синонимы в функции члене класса **cl** (если **mem** не был использован в качестве имени локальной переменной в промежуточной области видимости).

Функция член может быть определена в описании класса. Помещение определения функции члена в описание класса является кратким видом записи описания ее в описании класса и затем определения ее как **inline** сразу после описания класса.

Например:

```
int b;
struct x
   {
   int f () { return b; }
   int f () { return b; }
   int b;
   };
```

означает

```
int b;
struct x
   {
   int f ();
   int b;
   };
inline x.f () { return b; }
```

Для функций членов не нужно использование спецификатора **overload**: если имя описывается как означающее несколько имен в классе, то оно перегружено.

Применение операции получения адреса к функциям членам допустимо. Тип параметра результирующей функции указатель на есть **(...)**, то есть, неизвестен. Любое использование его является зависимым от реализации, поскольку способ

инициализации указателя для вызова функции члена не определен.

Производные классы

В конструкции

```
агрег идентификатор:public opt
typedef-имя
```

typedef-имя должно означать ранее описанный класс, называемый базовым классом для класса, подлежащего описанию. Говорится, что последний выводится из предшествующего.

На члены базового класса можно ссылаться, как если бы они были членами производного класса, за исключением тех случаев, когда имя базового члена было переопределено в производном классе; в этом случае для ссылки на скрытое имя может использоваться такая запись:

```
typedef-имя :: идентификатор
```

Например:

```
struct base
   {
   int a;
   int b;
   };
struct derived : public base
   {
   int b;
   int c;
   };
derived d;
d.a = 1;
d.base::b = 2;
d.b = 3;
d.c = 4;
```

осуществляет присваивание четырем членам **d**.

Производный тип сам может использоваться как базовый.

Виртуальные функции

Если базовый класс **base** содержит (виртуальную) **virtual** функцию **vf**, а производный класс **derived** также содержит

функцию **vf**, то вызов **vf** для объекта класса **derived** вызывает **derived::vf**.

Например:

```
struct base
   {
   virtual void vf ();
   void f ();
   };
struct derived : public base
   {
   void vf ();
   void f ();
   };
derived d;
base* bp = &d;
bp->vf ();
bp->f ();
```

Вызовы вызывают, соответственно, **derived::vf** и **base::f** для объекта класса **derived**, именованного **d**. Так что интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызвана, в то время как интерпретация вызова невиртуальной функции зависит только от типа указателя, обозначающего объект.

Из этого следует, что тип объектов классов с виртуальными функциями и объектов классов, выведенных из таких классов, могут быть определены во время выполнения.

Если производный класс имеет член с тем же именем, что и у виртуальной функции в базовом классе, то оба члена должны иметь одинаковый тип. Виртуальная функция не может быть другом (**friend**).

Функция **f** в классе, выведенном из класса, который имеет виртуальную функцию **f**, сама рассматривается как виртуальная. Виртуальная функция в базовом классе должна быть определена. Виртуальная функция, которая была определена в базовом классе, не нуждается в определении в производном классе.

В этом случае функция, определенная для базового класса, используется во всех вызовах.

Конструкторы

Член функция с именем, совпадающим с именем ее класса, называется конструктором. Конструктор не имеет типа возвращаемого значения; он используется для конструирования значений с типом его класса. С помощью конструктора можно создавать новые объекты его типа, используя синтаксис:

```
typedef-имя ( список_параметров opt )
```

Например:

```
complex zz = complex (1,2.3);
cprint (complex (7.8,1.2));
```

Объекты, созданные таким образом, не имеют имени (если конструктор не использован как инициализатор, как это было с **zz** выше), и их время жизни ограничено областью видимости, в которой они созданы. Они не могут рассматриваться как константы их типа.

Если класс имеет конструктор, то он вызывается для каждого объекта этого класса перед тем, как этот объект будет как-либо использован.

Конструктор может быть **overload**, но не **virtual** или **friend**. Если класс имеет базовый класс с конструктором, то конструктор для базового класса вызывается до вызова конструктора для производного класса.

Конструкторы для объектов членов, если таковые есть, выполняются после конструктора базового класса и до конструктора объекта, содержащего их.

Объяснение того, как могут быть специфицированы параметры для базового класса, а того, как конструкторы могут использоваться для управления свободной памятью.

Преобразования

Конструктор, получающий один параметр, определяет преобразование из типа своего параметра в тип своего класса. Такие преобразования неявно применяются дополнительно к обычным арифметическим преобразованиям.

Поэтому присваивание объекту из класса **X** допустимо, если или присваиваемое значение является **X**, или если **X** имеет конструктор, который получает присваиваемое значение как свой единственный параметр.

Аналогично конструкторы используются для преобразования параметров функции и инициализаторов.

Например:

```
class X { ... X (int); };
f (X arg)
   {
   X a = 1;
   /* a = X (1) */
   a = 2;
   /* a = X (2) */
   f (3);
   /* f (X (3)) */
   }
```

Если для класса **X** не найден ни один конструктор, принимающий присваиваемый тип, то не делается никаких попыток отыскать конструктор для преобразования присваиваемого типа в тип, который мог бы быть приемлем для конструкторов класса **X**.

Например:

```
class X { ... X (int); };
class X { ... Y (X); };
Y a = 1;
/* недопустимо: Y (X (1)) не пробуется */
```

Деструкторы

Функция член класса **cl** с именем **~cl** называется деструктором. Деструктор не возвращает никакого значения и не получает никаких параметров; он используется для уничтожения значений типа **cl** непосредственно перед уничтожением содержащего их объекта.

Деструктор не может быть **overload**, **virtual** или **friend**.

Деструктор для базового класса выполняется после деструктора производного от него класса. Как деструкторы используются для управления свободной памятью.

Видимость имен членов

Члены класса, описанные с ключевым словом **class**, являются закрытыми, это значит, что их имена могут использоваться только функциями членами и друзьями, пока они

не появятся после метки **public:**. В этом случае они являются общими.

Общий член может использоваться любой функцией. Структура является классом, все члены которого общие.

Если перед именем базового класса в описании производного класса стоит ключевое слово **public**, то общие члены базового класса являются общими для производного класса; если нет, то они являются закрытыми.

Общий член **mem** закрытого базового класса **base** может быть описан как общий для производного класса с помощью описания вида:

```
typedef-имя . идентификатор;
```

в котором **typedef**-имя означает базовый класс, а идентификатор есть имя члена базового класса. Такое описание может появляться в общей части производного класса.

Рассмотрим:

```
class base
   {
   int a;
   public:
   int b,c;
   int bf ();
   };
class derived : base
   {
   int d;
   public:
   base.c;
   int e;
   int df ();
   };
int ef (derived&);
```

Внешняя функция **ef** может использовать только имена **c**, **e** и **df**. Являясь членом **derived**, функция **df** может использовать имена **b**, **c**, **bf**, **d**, **e** и **df**, но не **a**. Являясь членом **base**, функция **bf** может использовать члены **a**, **b**, **c** и **bf**.

Друзья (friends)

Другом класса является функция **не-член**, которая может использовать имена закрытых членов. Следующий пример иллюстрирует различия между членами и друзьями:

```
class private
   {
   int a;
   friend void friend_set (private*,int);
   public:
   void member_set (int);
   };
void friend_set (private* p,int i)
{ p->a=i; }
void private.member_set (int i)
{ a = i; }
private obj;
friend_set (&obj,10);
obj.member_set (10);
```

Если описание **friend** относится к перегруженному имени или операции, то другом становится только функция с описанными типами параметров. Все функции класса **cl1** могут быть сделаны друзьями класса **cl2** с помощью одного описания

```
class cl2
   {
   friend cl1;
   . . .
   };
```

Функция-операция

Большинство операций могут быть перегружены с тем, чтобы они могли получать в качестве операндов объекты класса.

```
имя_функции_операции:   operator op
```

Где **op**:

- +
- -
- *
- /
- %

- ^
- &
- |
- ~
- !
- =
- <
- >
- +=
- -=
- *=
- /=
- %=
- ^=
- &=
- |=
- <<
- >>
- <<=
- >>=
- ==
- !=
- <=
- >=
- &&
- ||
- ++
- --
- ()
- []

Последние две операции — это вызов функции и индексирование. Функция операция может или быть функцией членом, или получать по меньшей мере один параметр класса.

Структуры

Структура есть класс, все члены которого общие. Это значит, что:

```
struct s { ... };
```

эквивалентно

```
class s { public: ... };
```

Структура может иметь функции члены (включая конструкторы и деструкторы).

Объединения

Объединение можно считать структурой, все объекты члены которой начинаются со смещения 0, и размер которой достаточен для содержания любого из ее объектов членов.

В каждый момент времени в объединении может храниться не больше одного из объектов членов.

Объединение может иметь функции члены (включая конструкторы и деструкторы).

Поля бит

Описатель члена вида:

```
идентификатор opt:  константное_выражение
```

определяет поле; его длина отделяется от имени поля двоеточием. Поля упаковываются в машинные целые; они не являются альтернативой слов. Поле, не влезающее в оставшееся в целом место, помещается в следующее слово. Поле не может быть шире слова.

На некоторых машинах они размещаются справа налево, а на некоторых слева направо. Неименованные поля полезны при заполнении для согласования внешне предписанных размещений (форматов).

В особых случаях неименованные поля длины 0 задают выравнивание следующего поля по границе слова. Не требуется аппаратной поддержки любых полей, кроме целых. Более того, даже целые поля могут рассматриваться как **unsigned**.

По этим причинам рекомендуется описывать поля как **unsigned**. К полям не может применяться операция получения адреса **&**, поэтому нет указателей на поля. Поля не могут быть членами объединения.

Вложенные классы

Класс может быть описан внутри другого класса. В этом случае область видимости имен внутреннего класса его и общих имен ограничивается охватывающим классом. За исключением этого ограничения допустимо, чтобы внутренний класс уже был описан вне охватывающего класса.

Описание одного класса внутри другого не влияет на правила доступа к закрытым членам и не помещает функции члены внутреннего класса в область видимости охватывающего класса.

Например:

```
int x;
class enclose /* охватывающий */
   {
   int x;
   class inner
          {
          int y;
          f () { x=1 }
          ...
          };
   g (inner*);
   ...
   };
int inner;  /* вложенный */
enclose.g (inner* p) { ... }
```

В этом примере **x** в **f** ссылается на **x**, описанный перед классом **enclose**. Поскольку y является закрытым членом **inner**, **g** не может его использовать. Поскольку **g** является членом **enclose**, имена, использованные в **g**, считаются находящимися в области видимости класса **enclose**.

Поэтому **inner** в описании параметров **g** относится к охваченному типу **inner**, а не к **int**.

Инициализация

Описание может задавать начальное значение описываемого идентификатора. Инициализатору предшествует =, и он состоит из выражения или списка значений, заключенного в фигурные скобки.

Синтаксис:

```
= expression
= { список_инициализаторов }
= { список_инициализаторов ,}
( список_выражений )
список_инициализаторов:
выражение список_инициализаторов,
список_инициализаторов
{ список_инициализаторов }
```

Все выражения в инициализаторе статической или внешней переменной должны быть константными выражениями или выражениями, которые сводятся к адресам ранее описанных переменных, возможно со смещением на константное выражение.

Автоматические и регистровые переменные могут инициализироваться любыми выражениями, включающими константы, ранее описанные переменные и функции.

Гарантируется, что неинициализированные статические и внешние переменные получают в качестве начального значения **«пустое значение»**. Когда инициализатор применяется к скаляру (указатель или объект арифметического типа), он состоит из одного выражения, возможно, заключенного в фигурные скобки.

Начальное значение объекта находится из выражения; выполняются те же преобразования, что и при присваивании.

Заметьте, что поскольку **()** не является инициализатором, то **«X a();»** является не описанием объекта класса **X**, а описанием функции, не получающей значений и возвращающей **X**.

Список инициализаторов

Когда описанная переменная является составной (класс или массив), то инициализатор может состоять из заключенного в фигурные скобки, разделенного запятыми списка

инициализаторов для членов составного объекта, в порядке возрастания индекса или по порядку членов.

Если массив содержит составные подобъекты, то это правило рекурсивно применяется к членам составного подобъекта. Если инициализаторов в списке меньше, чем членов в составном подобъекте, то составной подобъект дополняется нулями.

Фигурные скобки могут опускаться следующим образом. Если инициализатор начинается с левой фигурной скобки, то следующий за ней список инициализаторов инициализирует члены составного объекта; наличие числа инициализаторов, большего, чем число членов, считается ошибочным.

Если, однако, инициализатор не начинается с левой фигурной скобки, то из списка берутся только элементы, достаточные для сопоставления членам составного объекта, частью которого является текущий составной объект.

Например,

```
int x[] = { 1, 3, 5 };
```

описывает и инициализирует **x** как одномерный массив, имеющий три члена, поскольку размер не был указан и дано три инициализатора.

```
float y[4][3] =
   {
   { 1, 3, 5 },
   { 2, 4, 6 },
   { 3, 5, 7 }
   };
```

является полностью снабженной квадратными скобками инициализацией: 1,3 и 5 инициализируют первый ряд массива **y[0]**, а именно, **y[0][2]**.

Аналогично, следующие две строки инициализируют **y[1]** и **y[2]**. Инициализатор заканчивается раньше, поэтому **y[3]** инициализируется значением 0. В точности
тот же эффект может быть достигнут с помощью

```
float y[4][3] = { 1, 3, 5, 2, 4, 6, 3, 5, 7 };
```

Инициализатор для **y** начинается с левой фигурной скобки, но не начинается с нее инициализатор для **y[0]**, поэтому

используется три значения из списка. Аналогично, следующие три успешно используются для **y[1]** и следующие три для **y[2]**.

```
float y[4][3] = { { 1 }, { 2 }, { 3 }, { 4 } };
```

инициализирует первый столбец **y** (рассматриваемого как двумерный массив) и оставляет остальные элементы нулями.

Классовые объекты

Объект с закрытыми членами не может быть инициализирован с помощью простого присваивания, как это описывалось выше; это же относится к объекту объединение. Если класс имеет конструктор, не получающий значений, то этот конструктор используется для объектов, которые явно не инициализированы.

Параметры для конструктора могут также быть представлены в виде заключенного в круглые скобки списка.

Например:

```
struct complex
   {
   float re;
   float im;
   complex (float r,float i)
   { re=r; im=i; }
   complex (float r) { re=r; im=0; }
   };
complex zz (1,2.3);
complex* zp = new complex (1,2.3);
```

Инициализация может быть также выполнена с помощью явного присваивания; преобразования производятся.

Например:

```
complex zz1 = complex (1,2.3);
complex zz2 = complex (123);
complex zz3 = 123;
complex zz4 = zz3;
```

Если конструктор ссылается на объект своего собственного класса, то он будет вызываться при инициализации объекта другим объектом этого класса, но не при инициализации объекта конструктором.

Объект класса, имеющего конструкторы, может быть членом составного объекта только если он сам не имеет конструктора или если его конструкторы не имеют параметров. В последнем случае конструктор вызывается при создании составного объекта.

Если член составного объекта является членом класса с деструкторами, то этот деструктор вызывается при уничтожении составного объекта.

Ссылки

Когда переменная описана как **T&**, что есть **«ссылка на тип T»**, она может быть инициализирована или указателем на тип **T**, или объектом типа **T**. В последнем случае будет неявно применена операция взятия адреса **&**.

Например:

```
int i;
int& r1 = i;
int& r2 = &i;
```

И **r1** и **r2** будут указывать на **i**.

Обработка инициализации ссылки очень сильно зависит от того, что ей присваивается. Ссылка неявно переадресуется при ее использовании.

Например:

```
r1 = r2;
```

означает копирование целого, на которое указывает **r2**, в целое, на которое указывает **r1**.

Ссылка должна быть инициализирована. Таким образом, ссылку можно считать именем объекта.

Чтобы получить указатель **pp**, обозначающий тот объект, что и ссылка **rr**, можно написать **pp=&rr**. Это будет проинтерпретировано как:

```
pp=&*rr
```

Если инициализатор для ссылки на тип **T** не является адресным выражением, то будет создан и инициализирован с помощью правил инициализации объект типа **T**. Тогда значением ссылки станет адрес объекта. Время жизни объекта, созданного таким способом, будет в той области видимости, в которой он создан.

Например:

```
double& rr = 1;
```

допустимо, и **rr** будет указывать на объект типа **double**, в котором хранится значение 1.0.

Ссылки особенно полезны в качестве типов параметров.

Массивы символов

Последняя сокращенная запись позволяет инициализировать строкой массив данных типа **char**. В этом случае последовательные символы строки инициализируют члены массива.

Например:

```
char msg[] = "Syntax error on line %d\n";
```

демонстрирует массив символов, члены которого инициализированы строкой.

Имена типов

Иногда (для неявного задания преобразования типов и в качестве параметра **sizeof** или **new**) нужно использовать имя типа данных. Это выполняется при помощи **«имени типа»**, которое по сути является описанием для объекта этого типа, в котором опущено имя объекта.

Синтаксис:

```
спецификатор_типа абстрактный_описатель
Абстрактный_описатель:
пустой
*
абстрактный_описатель   абстрактный_описатель
( список_описателей_параметров)
абстрактный_описатель
[ константное_выражение opt ]
( абстрактный_описатель )
```

Является возможным идентифицировать положение в **абстрактном_описателе**, где должен был бы появляться идентификатор в случае, если бы конструкция была описателем в описании. Тогда именованный тип является тем же, что и тип предполагаемого идентификатора.

Например:

```
int
int *
int *[3]
int *()
int (*)()
```

именует, соответственно, типы **«целое»**, **«указатель на целое»**, **«указатель на массив из трех целых»**, **«функция, возвращающая указатель на функцию, возвращающую целое»** и **«указатель на целое»**.

Простое имя типа есть имя типа, состоящее из одного идентификатора или ключевого слова.

Простое_имя_типа:

- typedef-имя
- char
- short
- int
- long
- unsigned
- float
- double

Они используются в альтернативном синтаксисе для преобразования типов.

Например:

```
(double) a
```

может быть также записано как

```
double (a)
```

Определение типа typedef

Описания, содержащие **спецификатор_описания typedef**, определяют идентификаторы, которые позднее могут использоваться так, как если бы они были ключевыми словами типа, именующее основные или производные типы.

Синтаксис:

```
typedef-имя:
идентификатор
```

Внутри области видимости описания, содержащего **typedef**, каждый идентификатор, возникающий как часть какого-либо описателя, становится в этом месте синтаксически эквивалентным ключевому слову типа, которое именует тип, ассоциированный с идентификатором. Имя класса или перечисления также является **typedef**-именем.

Например, после

```
typedef int MILES, *KLICKSP;
struct complex { double re, im; };
```

каждая из конструкций

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

является допустимым описанием; **distance** имеет тип **int**, **metricp** имеет тип «**указатель на int**».

typedef не вводит новых типов, но только синонимы для типов, которые могли бы быть определены другим путем. Так в приведенном выше примере **distance** рассматривается как имеющая в точности тот же тип, что и любой другой **int** объект.

Но описание класса вводит новый тип.

Например:

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

описывает три переменных трех различных типов.

Описание вида:

```
агрег                    идентификатор ;
enum идентификатор ;
```

определяет то, что идентификатор является именем некоторого (возможно, еще не определенного) класса или перечисления. Такие описания позволяют описывать классы, ссылающихся друг на друга.

Например:

```
class vector;
class matrix
   {
   ...
   friend matrix operator* (matrix&,vector&);
   };
class vector
   {
   ...
   friend matrix operator*
   (matrix&,vector&);
   };
```

Перегруженные имена функций

В тех случаях, когда для одного имени определено несколько (различных) описаний функций, это имя называется перегруженным.

При использовании этого имени правильная функция выбирается с помощью сравнения типов фактических параметров с типами параметров в описаниях функций. К перегруженным именам неприменима операция получения адреса **&**.

Из обычных арифметических преобразований для вызова перегруженной функции выполняются только **char->short->int**, **int->double**, **int->long** и **float->double**.

Для того, чтобы перегрузить имя функции **не-члена** описание **overload** должно предшествовать любому описанию функции.

Например:

```
overload abs;
int abs (int);
double abs (double);
```

Когда вызывается перегруженное имя, по порядку производится сканирование списка функций для нахождения той, которая может быть вызвана.

Например, **abs(12)** вызывает **abs(int)**, а **abs(12.0)** будет вызывать **abs(double)**. Если бы был зарезервирован порядок вызова, то оба обращения вызвали бы **abs(double)**.

Если в случае вызова перегруженного имени с помощью вышеуказанного метода не найдено ни одной функции, и если функция получает параметр типа класса, то конструкторы классов параметров (в этом случае существует единственный набор преобразований, делающий вызов допустимым) применяются неявным образом.

Например:

```
class X { ... X (int); };
class Y { ... Y (int); };
class Z { ... Z (char*); };
overload int f (X), f (Y);
overload int g (X), g (Y);
f (1);
/* неверно: неоднозначность f(X(1))  или f(Y(1)) */
g (1); /* g(X(1)) */
g ("asdf"); /* g(Z("asdf")) */
```

Все имена функций операций являются автоматически перегруженными.

Описание перечисления

Перечисления являются **int** с именованными константами.

```
enum_спецификатор:
enum идентификатор opt { enum_список }
enum_список:
перечислитель
enum_список, перечислитель
Перечислитель:
идентификатор
идентификатор = константное_выражение
```

Идентификаторы в **enum**-списке описаны как константы и могут появляться во всех местах, где требуются константы.

Если не появляется ни одного перечислителя **с =**, то значения всех соответствующих констант начинаются с 0 и возрастают на 1 по мере чтения описания слева направо.

Перечислитель **с =** дает ассоциированному с ним идентификатору указанное значение; последующие идентификаторы продолжают прогрессию от присвоенного значения.

Имена перечислителей должны быть отличными от имен обычных переменных. Значения перечислителей не обязательно должны быть различными.

Роль идентификатора в спецификаторе перечисления **enum_спецификатор** полностью аналогична роли имени класса; он именует определенный нумератор.

Например:

```
enum color { chartreuse, burgundy, claret=20, winedark };
   ...
   color *cp, col;
   ...
   col = claret;
   cp = &col;
   ...
   if (*cp == burgundy) ...
```

делает **color** именем типа, описывающего различные цвета, и затем описывает cp как указатель на объект этого типа. Возможные значения извлекаются из множества **{ 0, 1, 20, 21 }**.

Описание Asm

Описание **Asm** имеет вид:

```
asm (строка);
```

Смысл описания **asm** не определен. Обычно оно используется для передачи информации ассемблеру через компилятор.

Операторы

Операторы выполняются последовательно во всех случаях кроме особо оговоренных.

Оператор выражение

Большинство операторов является операторами выражение, которые имеют вид

```
выражение ;
```

Обычно операторы выражение являются присваиваниями и вызовами функций.

Составной оператор, или блок

Составной оператор (называемый также «**блок**», что эквивалентно) дает возможность использовать несколько операторов в том месте, где предполагается использование одного:

```
Составной_оператор:
{ список_описаний opt список_операторов opt }
список_описаний:
описание
описание список_описаний
Список_операторов:
оператор
оператор список_операторов
```

Если какой-либо из идентификаторов в **списке_описаний** был ранее описан, то внешнее описание выталкивается на время выполнения блока, и снова входит в силу по его окончании.

Каждая инициализация **auto** или **register** переменных производится всякий раз при входе в голову блока. В блок делать передачу; в этом случае инициализации не выполняются. Инициализации переменных, имеющих класс памяти **static** осуществляются только один раз в начале выполнения программы.

Условный оператор

Есть два вида условных операторов:

```
if ( выражение ) оператор
if ( выражение ) оператор else оператор
```

В обоих случаях вычисляется выражение, и если оно не ноль, то выполняется первый подоператор. Во втором случае второй подоператор выполняется, если выражение есть 0. Как обычно, неоднозначность «**else**» разрешается посредством того, что **else** связывается с последним встреченным **if**, не имеющим **else**.

Оператор while

Оператор **while** имеет вид:

```
while ( выражение ) оператор
```

Выполнение подоператора повторяется, пока значение выражения остается ненулевым. Проверка выполняется перед каждым выполнением оператора.

Оператор do

Оператор **do** имеет вид:

```
do оператор while (выражение);
```

Выполнение подоператора повторяется до тех пор, пока значение выражения не станет нулем. Проверка выполняется после каждого выполнения оператора.

Оператор for

Оператор **for** имеет вид:

```
for ( выражение_1 opt ;
выражение_2 opt ;
выражение_3 opt )
оператор
```

Этот оператор эквивалентен следующему:

```
выражение_1;
while (выражение_2)
{ оператор выражение_3; }
```

- первое выражение задает инициализацию цикла;
- второе выражение задает осуществляемую перед каждой итерацией проверку, по которой производится выход из цикла, если выражение становится нулем;
- третье выражение часто задает приращение, выполняемое после каждой итерации.

Каждое или все выражения могут быть опущены. Отсутствие **выражения_2** делает подразумеваемое **while**-предложение эквивалентным **while(1)**; остальные опущенные выражения просто пропускаются в описанном выше расширении.

Оператор switch

Оператор **switch** вызывает передачу управления на один из нескольких операторов в зависимости от значения выражения. Он имеет вид

```
switch ( выражение ) оператор
```

Выражение должно быть целого типа или типа указателя. Любой оператор внутри оператора может быть помечен одним или более префиксом **case** следующим образом:

```
case константное_выражение :
```

где **константное_выражение** должно иметь тот же тип что и выражение-переключатель; производятся обычные арифметические преобразования. В одном операторе **switch** никакие две константы, помеченные **case**, не могут иметь одинаковое значение.

Может также быть не более чем один префикс оператора вида

```
default :
```

Когда выполнен оператор **switch**, проведено вычисление его выражения и сравнение его с каждой **case** константой.

Если одна из констант равна значению выражения, то управление передается на выражение, следующее за подошедшим префиксом **case**.

Если никакая **case** константа не соответствует выражению, и есть префикс **default**, то управление передается на выражение, которому он предшествует.

Если нет соответствующих вариантов **case** и **default** отсутствует, то никакой из операторов в операторе **switch** не выполняется.

Префиксы **case** и **default** сами по себе не изменяют поток управления, который после задержки идет дальше, перескакивая через эти префиксы.

Обычно зависящий от **switch** оператор является составным. В голове этого оператора могут стоять описания, но инициализации автоматических и регистровых переменных являются безрезультатными.

Оператор break

Оператор

```
break ;
```

прекращает выполнение ближайшего охватывающего **while**, **do**, **for** или **switch** оператора; управление передается на оператор, следующий за законченным.

Оператор continue

Оператор

```
continue ;
```

вызывает передачу управления на управляющую продолжением цикла часть наименьшего охватывающего оператора **while**, **do** или **for**; то есть на конец петли цикла. Точнее, в каждом из операторов

```
while (...)
do
for (...)
{
   {
          {
          ...
          ...
          ...
          contin:;
          contin:;
          contin:;
          }
   }
}
while (...);
```

continue эквивалентно **goto contin** (За **contin:** следует пустой оператор).

Оператор return

Возврат из функции в вызывающую программу осуществляется с помощью оператора **return**, имеющего один из двух видов:

```
return ;
return выражение ;
```

Первый может использоваться только в функциях, не возвращающих значения, т.е. в функциях с типом возвращаемого значения **void**.

Вторая форма может использоваться только в функциях, не возвращающих значение; вызывающей функцию программе возвращается значение выражения.

Если необходимо, то выражение преобразуется, как это делается при присваивании, к типу функции, в которой оно возникло.

Обход конца функции эквивалентен возврату **return** без возвращаемого значения.

Оператор goto

Можно осуществлять безусловную передачу управления с помощью оператора

```
goto идентификатор ;
```

Идентификатор должен быть меткой, расположенной в текущей функции.

Помеченные операторы

Перед любым оператором может стоять префикс метка, имеющий вид

```
идентификатор :
```

который служит для описания идентификатора как метки. Метка используется только как объект для **goto**.

Областью видимости метки является текущая функция, исключая любой подблок, в котором был переописан такой же идентификатор.

Пустой оператор

Пустой оператор имеет вид

```
;
```

Пустой оператор используется для помещения метки непосредственно перед **}** составного оператора или того, чтобы снабдить такие операторы, как **while**, пустым телом.

Оператор delete

Оператор **delete** имеет вид

```
delete выражение ;
```

Результатом выражения должен быть указатель. Объект, на который он указывает, уничтожается. Это значит, что после оператора уничтожения **delete** нельзя гарантировать, что объект имеет определенное значение.

Эффект от применения **delete** к указателю, не полученному из операции **new**, не определен. Однако, уничтожение указателя с нулевым значением безопасно.

Оператор asm

Оператор **asm** имеет вид

```
asm ( строка ) ;
```

Смысл оператора **asm** не определен. Обычно он используется для передачи информации через компилятор ассемблеру.

Внешние определения

Программа на **C++** состоит из последовательности внешних определений. Внешнее определение описывает идентификатор как имеющий класс памяти **static** и определяет его тип. Спецификатор типа может также быть пустым, и в этом случае принимается тип **int**.

Область видимости внешних определений простирается до конца файла, в котором они описаны, так же, как действие описаний сохраняется до конца блока. Синтаксис внешних определений тот же, что и у описаний, за исключением того, что только на этом уровне и внутри описаний классов может быть задан код (текст программы) функции.

Определения функций

Определения функций имеют вид:

```
определение_функции:
спецификаторы_описания описатель_функции opt
инициализатор_базового_класса
opt тело_функции
```

Единственными спецификаторами класса памяти (**sc-спецификаторами**), допустимыми среди спецификаторов описания, являются **extern**, **static**, **overload**, **inline** и **virtua**l.

Описатель функции похож на описатель **«функции, возвращающей ...»**, за исключением того, что он включает в себя имена формальных параметров определяемой функции.

Описатель функции имеет вид:

```
описатель_функции:
описатель ( список_описаний_параметров )
```

Единственный класс памяти, который может быть задан, это тот, при котором соответствующий фактический параметр будет скопирован, если это возможно, в регистр при входе в функцию. Если в качестве инициализатора для параметра задано константное выражение, то это значение используется как значение параметра по умолчанию.

Тело функции имеет вид

```
тело_функции:
составной_оператор
```

Вот простой пример полного определения функции:

```
int max (int a,int b,int c)
   {
   int m = (a > b) ? a : b;
   return  (m > c) ? m : c;
   }
```

Здесь

- **int** является спецификатором типа;
- **max (int a, int b, int c)** является описателем функции;
- **{ ... }** — блок, задающий текст программы (код) оператора.

Поскольку в контексте выражения имя (точнее, имя как формальный параметр) считается означающим указатель на первый элемент массива, то описания формальных параметров, описанных как **«массив из ...»**, корректируются так, чтобы читалось **«указатель на ...»**.

Инициализатор базового класса имеет вид:

```
инициализатор_базового_класса:
: ( список_параметров opt )
```

Он используется для задания параметров конструктора базового класса в конструкторе производного класса.

Например:

```
struct base { base (int); ... };
struct derived : base
{ derived (int); ... };
```

```
derived.derived (int a) : (a+1)
{ ... }
derived d (10);
```

Конструктор базового класса вызывается для объекта **d** с параметром 11.

Определения внешних данных

Определения внешних данных имеют вид:

```
определение_данных:
описание
```

Класс памяти таких данных статический.

Если есть более одного определения внешних данных одного имени, то определения должны точно согласовываться по типу и классу памяти, и инициализаторы (если они есть), должны иметь одинаковое значение.

Командные строки компилятора

Компилятор языка **C++** содержит препроцессор, способный выполнять макроподстановки, условную компиляцию и включение именованных файлов. Строки, начинающиеся с #, относятся к препроцессору. Эти строки имеют независимый от остального языка синтаксис; они могут появляться в любом месте оказывать влияние, которое распространяется (независимо от области видимости) до конца исходного файла программы.

Заметьте, что определения **const** и **inline** дают альтернативы для большинства использований **#define**.

Замена идентификаторов

Командная строка компилятора имеет вид:

```
#define идент строка_символов
```

и вызывает замену препроцессором последующих вхождений идентификатора, заданного строкой символов. Точка с запятой внутри (или в конце) строки символов является частью этой строки.

Строка:

```
#define идент ( идент , ..., идент ) строка_символов
```

где отсутствует пробел между первым идентификатором и скобка, является макроопределением с параметрами. Последующие

вхождения первого идентификатора с идущими за ним **(**, последовательностью символов, разграниченной запятыми, и **)**, заменяются строкой символов, заданной в определении.

Каждое местоположение идентификатора, замеченного в списке параметров определения, заменяется соответствующей строкой из вызова. Фактическими параметрами вызова являются строки символов, разделенные запятыми; однако запятые в строке, заключенной в кавычки, или в круглых скобках не являются разделителями параметров.

Число формальных и фактических параметров должно совпадать. Строки и символьные константы в символьной строке сканируются в поисках формальных параметров, но строки и символьные константы в остальной программе не сканируются в поисках определенных (с помощью **define**) идентификаторов.

В обоих случаях строка замещения еще раз сканируется в поисках других определенных идентификаторов. В обоих случаях длинное определение может быть продолжено на другой строке с помощью записи \ в конце продолжаемой строки.

Командная строка:

```
#undef идент
```

влечет отмену препроцессорного определения идентификатора.

Включение файлов

Командная строка компилятора:

```
#include "имя_файла"
```

вызывает замену этой строки полным содержимым файла **имя_файла**. Сначала именованный файл ищется в директории первоначального исходного файла, а затем в стандартных или заданных местах.

Альтернативный вариант. Командная строка:

```
#include <имя_файла>
```

производит поиск только в стандартном или заданном месте, и не ищет в директории первоначального исходного файла. (То, как эти места задаются, не является частью языка.)

Включения с помощью **#include** могут быть вложенными.

Условная компиляция

Командная строка компилятора:

```
#if выражение
```

проверяет, является ли результатом вычисления выражения **не-ноль**. Выражение должно быть константным выражением. Применительно к использованию данной директивы есть дополнительные ограничения: константное выражение не может содержать **sizeof** или перечислимые константы.

Кроме обычных операций **Си** может использоваться унарная операция **defined**. В случае применения к идентификатору она дает значение **не-ноль**, если этот идентификатор был ранее определен с помощью **#define** и после этого не было отмены определения с помощью **#undef**; иначе ее значение **0**.

Командная строка:

```
#ifdef идент
```

проверяет, определен ли идентификатор в препроцессоре в данный момент; то есть, был ли он объектом командной строки **#define**.

Командная строка:

```
#ifndef идент
```

проверяет, является ли идентификатор неопределенным в препроцессоре в данный момент.

После строки каждого из трех видов может стоять произвольное количество строк, возможно, содержащих командную строку

```
#else
```

и далее до командной строки

```
#endif
```

Если проверенное условие истинно, то все строки между **#else** и **#endif** игнорируются. Если проверенное условие ложно, то все строки между проверкой и **#else** или, в случае отсутствия **#else**, **#endif**, игнорируются.

Эти конструкции могут быть вложенными.

Управление строкой

Для помощи другим препроцессорам, генерирующим программы на Си.

Строка:

```
#line константа "имя_файла"
```

заставляет компилятор считать, например, в целях диагностики ошибок, что константа задает номер следующей строки исходного файла, и текущий входной файл именуется идентификатором. Если идентификатор отсутствует, то запомненное имя файла не изменяется.

Обзор типов

Здесь кратко собрано описание действий, которые могут совершаться над объектами различных типов.

Классы

Классовые объекты могут присваиваться, передаваться функциям как параметры и возвращаться функциями. Другие возможные операции, как, например, проверка равенства, могут быть определены пользователем.

Функции

Есть только две вещи, которые можно проделывать с функцией: вызывать ее и брать ее адрес. Если в выражении имя функции возникает не в положении имени функции в вызове, то генерируется указатель на функцию. Так, для передачи одной функции другой можно написать

```
typedef int (*PF) ();
extern g (PF);
extern f ();
...
g (f);
```

Тогда определение **g** может иметь следующий вид:

```
g (PF funcp)
   {
   ...
   (*funcp) ();
   ...
   }
```

Заметьте, что **f** должна быть описана явно в вызывающей программе, поскольку ее появление в **g(f)** не сопровождалось **(**.

Массивы, указатели и индексирование

Всякий раз, когда в выражении появляется идентификатор типа массива, он преобразуется в указатель на первый член массива. Из-за преобразований массивы не являются адресами. По определению операция индексирования **[]** интерпретируется таким образом, что **E1[E2]** идентично ***((E1)+(E2))**.

В силу правил преобразования, применяемых к **+**, если **E1** массив и **E2** целое, то **E1[E2]** относится к **E2**-ому члену **E1**.

Поэтому, несмотря на такое проявление асимметрии, индексирование является коммутативной операцией.

Это правило сообразным образом применяется в случае многомерного массива. Если **E** является **n**-мерным массивом ранга **i*j*...*k**, то возникающее в выражении **E** преобразуется в указатель на **(n-1)**-мерный массив ранга **j*...*k**.

Если к этому указателю, явно или неявно, как результат индексирования, применяется операция *, ее результатом является **(n-1)**-мерный массив, на который указывалось, который сам тут же преобразуется в указатель.

Например:

```
int x[3][5];
```

Здесь **x** — массив целых размером 3 на 5. Когда **x** возникает в выражении, он преобразуется в указатель на (первый из трех) массив из 5 целых. В выражении **x[i]**, которое эквивалентно ***(x+1)**, **x** сначала преобразуется, как описано, в указатель, затем 1 преобразуется к типу **x**, что включает в себя умножение 1 на длину объекта, на который указывает указатель, а именно объект из 5 целых.

Результаты складываются, и используется косвенная адресация для получения массива (из 5 целых), который в свою очередь преобразуется в указатель на первое из целых.

Если есть еще один индекс, снова используется тот же параметр; на этот раз результат является целым.

Именно из всего этого проистекает то, что массивы в Си хранятся по строкам (быстрее всего изменяется последний индекс), и что в описании первый индекс помогает определить объем памяти, поглощаемый массивом, но не играет никакой другой роли в вычислениях индекса.

Явные преобразования указателей

Определенные преобразования, включающие массивы, выполняются, но имеют зависящие от реализации аспекты. Все они задаются с помощью явной операции преобразования типов.

Указатель может быть преобразован к любому из целых типов, достаточно больших для его хранения. То, какой из **int** и **long** требуется, является машинно-зависимым. Преобразующая функция также является машинно-зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине.

Объект целого типа может быть явно преобразован в указатель. Преобразующая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машинно-зависимой.

Указатель на один тип может быть преобразован в указатель на другой тип. Использование результирующего указателя может вызывать особые ситуации, если исходный указатель не указывает на объект, соответствующим образом выровненный в памяти.

Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений.

Например, программа, выделяющая память, может получать размер (в байтах) размещаемого объекта и возвращать указатель на **char**; это можно использовать следующим образом.

```
extern void* alloc ();
double* dp;
dp = (double*) alloc (sizeof double));
*dp= 22.0 / 7.0;
```

alloc должна обеспечивать (машинно-зависимым образом) то, что возвращаемое ею значение подходит для преобразования в указатель на **double**; в этом случае использование функции мобильно. Различные машины различаются по числу бит в указателях и требованиям к выравниванию объектов. Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

Константные выражения

В нескольких местах С++ требует выражения, вычисление которых дает константу: в качестве границы массива, в **case** выражениях, в качестве значений параметров функции, присваиваемых по умолчанию и в инициализаторах.

В первом случае выражение может включать только целые константы, символьные константы, константы, описанные как имена, и **sizeof** выражения, возможно, связанные бинарными операциями:

- +
- -
- *
- /
- %
- &
- |
- ^
- <<
- >>
- ==
- !=
- <
- >
- <=
- >=
- &&
- ||

или унарными операциями:

- -
- ~
- !

или тернарными операциями:

- ?
- :

Скобки могут использоваться для группирования, но не для вызова функций.

Большая широта допустима для остальных трех случаев использования; помимо константных выражений, обсуждавшихся выше, допускаются константы с плавающей точкой, и можно также применять унарную операцию **&** к внешним или статическим объектам, или к внешним или статическим массивам, индексированным константным выражением. Унарная операция **&** может также быть применена неявно с помощью употребления неиндексированных массивов и функций.

Основное правило состоит в том, что инициализаторы должны при вычислении давать константу или адрес ранее описанного внешнего или статического объекта плюс или минус константа.

Меньшая широта допустима для константных выражений после **#if:** константы, описанные как имена, **sizeof** выражения и перечислимые константы недопустимы.

Соображения мобильности

Определенные части **C++** являются машинно-зависимыми по своей сути.

Как показала практика, характеристики аппаратуры в чистом виде, такие, как размер слова, свойства плавающей арифметики и целого деления, не создают особых проблем. Другие аппаратные аспекты отражаются на различных программных разработках.

Некоторые из них, особенно знаковое расширение (преобразование отрицательного символа в отрицательное целое) и порядок расположения байтов в слове, являются досадными помехами, за которыми надо тщательно следить. Большинство других являются всего лишь мелкими сложностями.

Число регистровых переменных, которые фактически могут быть помещены в регистры, различается от машины к машине,

как и множество фактических типов. Тем не менее, все компиляторы на «своей» машине все делают правильно; избыточные или недействующие описания **register** игнорируются.

Некоторые сложности возникают при использовании двусмысленной манеры программирования. Писать программы, зависящие от какой-либо из этих особенностей, районе неблагоразумно.

В языке не определен порядок вычисления параметров функции. На некоторых машинах он слева направо, а на некоторых справа налево.

Порядок появления некоторых побочных эффектов также недетерминирован.

Поскольку символьные константы в действительности являются объектами типа **int**, то могут быть допустимы многосимвольные константы. Однако конкретная реализация очень сильно зависит от машины, поскольку порядок, в котором символы присваиваются слову, различается от машины к машине. На некоторых машинах поля в слове присваиваются слева направо, на других справа налево.

Эти различия невидны для отдельных программ, не позволяющих себе каламбуров с типами (например, преобразования **int** указателя в **char** указатель и просмотр памяти, на которую указывает указатель), но должны приниматься во внимание при согласовании внешне предписанных форматов памяти.

Свободная память

Операция **new** вызывает функцию

```
extern void* _new (long);
```

для получения памяти. Параметр задает число требуемых байтов. Память будет инициализирована. Если **_new** не может найти требуемое количество памяти, то она возвращает ноль.

Операция **delete** вызывает функцию

```
extern void _delete (void*);
```

чтобы освободить память, указанную указателем, для повторного использования. Результат вызова **_delete()** для указателя, который не был получен из **_new()**, не определен, это же относится и к

повторному вызову **_delete()** для одного и того же указателя. Однако уничтожение с помощью **delete** указателя со значением ноль безвредно.

Предоставляются стандартные версии **_new()** и **_delete()**, но пользователь может применять другие, более подходящие для конкретных приложений.

Когда с помощью операции **new** создается классовый объект, то для получения необходимой памяти конструктор будет (неявно) использовать **new**.

Конструктор может осуществить свое собственное резервирование памяти посредством присваивания указателю **this** до каких-либо использований.

С помощью присваивания **this** значения ноль деструктор может избежать стандартной операции дерезервирования памяти для объекта его класса.

Например:

```
class cl
   {
   int v[10];
   cl () { this = my_own_allocator
   (sizeof (cl)); }
   ~cl () { my_own_deallocator
   (this); this = 0; }
   }
```

На входе в конструктор **this** является **не-нулем**, если резервирование памяти уже имело место (как это имеет место для автоматических объектов), и **нулем** в остальных случаях.

Если производный класс осуществляет присваивание **this**, то вызов конструктора (если он есть) базового класса будет иметь место после присваивания, так что конструктор базового класса ссылаться на объект посредством конструктора производного класса.

Если конструктор базового класса осуществляет присваивание **this**, то значение также будет использоваться конструктором (если таковой есть) производного класса.

Справочник по работе с DOS

Управление памятью в DOS

Нехватка памяти при выполнении

Borland C++ при компиляции не генерирует на диске никаких промежуточных структур данных (записывая на диск только файлы **.OBJ**). Вместо этого для хранения промежуточных структур данных между проходами используется оперативная память. Поэтому при недостаточном объеме оперативной памяти вам может выводиться сообщение о нехватке памяти. Чтобы решить эту проблему, уменьшите размер функций или разбейте файл с крупными функциями на несколько частей.

Модели памяти

В Borland C++ используется 6 моделей памяти, каждая из которых служит для различных размеров программ и кода.

Регистры процессора

Ниже представлены некоторые регистры процессоров. Эти процессоры имеют и другие регистры, но непосредственно к ним обращаться нельзя, поэтому они здесь не показаны.

Регистры общего назначения

Аккумулятор (математические операции)

AX AH AL

Базовый регистр (индексирование)

BX BH BL

Счетчик (индексирование)

CX CH CL

Регистр данных

DX DH DL

Сегментные адресные регистры

CS Сегментный регистр кода

DS Сегментный регистр данных

SS Указатель сегмента стека

ES Дополнительный регистр сегмента

Регистры общего назначения

SP Указатель стека

BP Указатель базы

SI Индекс источника

DI Индекс приемника

Общие регистры чаще всего используются для работы с данными. Каждый из них выполняет некоторые специальные функции, которые доступны только ему, например, некоторые математические операции могут использовать только регистр **AX**, регистр **BX** может служить базовым регистром, **CX** применяется инструкцией **LOOP** и некоторыми строковыми инструкциями, а **DX** используется некоторыми математическими операциями неявно. Однако во многих операциях можно использовать все эти регистры и заменять один из них на другой.

Сегментные регистры содержат начальный адрес каждого из 4 сегментов. 16-разрядное значение в сегментном регистре для получения 20-разрядного адреса сегмента сдвигается влево на 4 (умножается на 16).

Процессоры имеют также некоторые специальные регистры:

- Регистры **SI** и **DI** могут выполнять многие функции общих регистров, но могут также использоваться в качестве индексных регистров. Они используются и в регистровых переменных Borland C++.
- Регистр **SP** указывает на текущую вершину стека и представляет смещение в сегменте стека.
- Регистр **BP** — это вспомогательный указатель стека, применяемый для индексирования в стеке с целью извлечения аргументов или локальных динамических переменных.

Функции Borland C++ используют регистр базы (**BP**) в качестве базового регистра для аргументов и переменных. Параметры имеют положительные смещения от **BP**, зависящие от модели памяти. При наличии кадра стека **BP** указывает на

сохраненное предыдущее значение **BP**. Если параметр **Standard Stack Frame** выключен (**Off**), то функции без аргументов не используют и не сохраняют **BP**.

16-разрядный регистр флагов содержит все необходимую информацию о состоянии процессора и результатах последних инструкций.

V

Виртуальный режим

R

Возобновление

N

Вложенная задача

IOP

Уровень защиты ввода-вывода

O

Переполнение

D

Направление

I

Разрешение прерывания

T

Прерывание

S

Знак

Z

Признак нуля

A

Вспомогательный перенос

P

Четность

C

Перенос

Например, если вы хотите знать, получен ли при вычитании нулевой результат, непосредственно после этой инструкции вам следует проверить флаг нуля (бит **Z** в регистре флагов). Если он установлен (то есть имеет ненулевое значение), это будет говорить о том, что результат нулевой. Другие флаги, такие, как флаги переноса и переполнения аналогичным образом сообщают о результатах арифметических и логических операций.

Прочие флаги управляют режимом операций процессора. Флаг направления управляет направлением, в котором строковые инструкции выполняют перемещение, а флаг прерывания управляет тем, будет ли разрешено внешним аппаратным средствам, таким, например, как клавиатура или модем, временно приостанавливать текущий код для выполнения функций, требующих немедленного обслуживания. Флаг перехвата используется только программным обеспечением, которое служит для отладки другого программного обеспечения (отладчики).

Регистр флагов не считывается и не модифицируется непосредственно. Вместо этого регистр флагов управляется в общем случае с помощью специальных инструкций (таких, как **CLD**, **STI** и **CMC**), а также с помощью арифметических и логических инструкций, модифицирующих отдельные флаги. И наоборот, содержимое отдельных разрядов регистра флагов влияет на выполнение инструкций (например, **JZ**, **RCR** и **MOVSB**). Регистр флагов не используется на самом деле, как ячейка памяти, вместо этого он служит для контроля за состоянием и управления процессором.

Сегментация памяти

Память микропроцессора Intel имеет сегментированную архитектуру. Непосредственно можно адресоваться к 64K памяти сегменту. Процессор отслеживает 4 различных сегмента: сегмент кода, сегмент данных, сегмент стека и дополнительный сегмент. В сегменте кода находятся машинные инструкции, а в дополнительном сегменте — дополнительные данные. Процессор имеет 4 16-разрядных сегмента (по одному на сегмент) — **CS**, **DS**, **SS** и **ES**, которые указывают на сегмент кода, данных, стека и дополнительный сегмент соответственно. Сегмент может находиться в любом месте памяти, но начинаться должен по адресу, кратному 10. Сегменты могут перекрываться. Например, все четыре сегмента могут начинаться с одного адреса.

Стандартная запись адреса имеет форму «**сегмент:смещение**», например, **2F84:0546**. Начальный адрес сегмента всегда представляет собой 20-битовое число, но так как сегментный регистр содержит только 16 бит, нижние 4 бита полагаются равными 0. Это значит, что сегменты могут начинаться только с тех адресов, у которых последние 4 бита равны 0.

Указатели

Хотя указатель или функция могут иметь конкретный тип независимо от используемой модели, вы можете выбрать заданный по умолчанию тип указателя, используемый для кода и данных. Существует 4 типа указателей:

- **near** (16 бит)
- **far** (32 бита)
- **huge** (32 бита)
- **segment** (16 бит).

В указателях **near** (ближние указатели) для вычисления адреса используется один сегментный регистр, например, 16-битовое значение указателя функции складывается со сдвинутым влево содержимым регистра кода **CS**. С такими указателями легко работать.

Указатели **far** (дальние указатели) содержат не только смещение в сегменте, но и адрес сегмента (другое 16-битовое значение). Такие указатели позволяют иметь несколько сегментов кода и программы, превышающие по размеру 64K. Здесь нужно учитывать, что в операциях **==** и **!=** используются 32-битовые значения **unsigned long**, а не полный адрес памяти. В операциях сравнения <=, >=, < и > используется только смещение.

При прибавлении к указателю значения изменяется только смещение. Если смещение превышает **FFFF** (максимально возможное значение), то указатель возвращается к началу сегмента. При сравнении указателей лучше использовать ближние указатели или указатели **huge**.

Указатели **huge** также занимают 32 бита. Аналогично указателям **far**, они содержат и адрес сегмента и смещение. Однако, чтобы избежать проблем с указателями, такие указатели нормализуются. Нормализованный указатель — это 32-битовый указатель с максимально возможным значением в сегментном

адресе. Так как сегмент может начинаться с каждых 16 байт, это означает, что данное смещение будет иметь значение от 0 до 15. Для нормализации указателя он конвертируется в 20-битовый адрес, а затем используются правые 4 бита смещения и левые 16 бит адреса сегмента. Например, **2F84:0532** преобразуется в абсолютный адрес **2FD72**, который нормализуется в **2FD7:0002**. Нормализация важна по следующими причинам:

- Каждому сегментному адресу соответствует при этом только одна возможная адресная пара «**сегмент:смещение**». Это означает, что операции **==** и **!=** возвращают корректный ответ.
- В операциях сравнения <=, >=, < и > используется при этом полные 32-битовые значения. Нормализация обеспечивает корректность результатов.
- Благодаря нормализации смещение в указателях **huge** автоматически циклически возвращаются каждые 16 байт, но настраивается также и сегмент. Например, при инкрементации **811B:000F** результатом будет **811C:0000**. Это обеспечивает, что, например, при наличии массива структур типа **huge** > **64K** индексирование массива и выбор поля **struct** будет работать для структур любого размера.

Однако работа с указателями **huge** связана с дополнительными издержками. Из-за этого арифметические операции с указателями **huge** выполняются намного медленнее, чем с указателями **far**.

Модели памяти

В 16-разрядных программах Borland C++ вы можете использовать 6 моделей памяти: крохотную, малую, среднюю, компактную, большую и огромную.

Tiny (крохотная)

Эта модель памяти используется в тех случаях, когда абсолютным критерием достоинства программы является размер ее загрузочного кода. Это минимальная из моделей памяти. Все четыре сегментных регистра (**CS**, **DS**, **SS** и **ES**) устанавливаются на один и тот же адрес, что дает общий размер кода, данных и стека, равный 64K. Используются исключительно ближние

указатели. Программы со сверхмалой моделью памяти можно преобразовать к формату **.COM** (при компоновке с параметром **/t**).

Small (малая)

Эта модель хорошо подходит для небольших прикладных программ. Сегменты кода и данных расположены отдельно друг от друга и не перекрываются, что позволяет иметь 64K кода программы и 64K данных и стека. Используются только указатели **near**.

Medium (средняя)

Эта модель годится для больших программ, для которых не требуется держать в памяти большой объем данных. Для кода, но не для данных используются указатели **far**. В результате данные плюс стек ограничены размером 64K, а код может занимать до 1M.

Compact (компактная)

Лучше всего использовать эту модель в тех случаях, когда размер кода невелик, но требуется адресация большого объема данных. Указатели **far** используются для данных, но не для кода. Следовательно, код здесь ограничен 64K, а предельный размер данных — 1 Мб.

Large (большая)

Модели **large** и **huge** применяются только в очень больших программах. Дальние указатели используются как для кода, так и для данных, что дает предельный размер 1 Мб для обоих.

Huge (огромная)

Дальние указатели используются как для кода, так и для данных. Borland C++ обычно ограничивает размер статических данных 64K; модель памяти **huge** отменяет это ограничение, позволяя статическим данным занимать более 64K.

Для выбора любой из этих моделей памяти вы должны либо воспользоваться соответствующим параметром меню интегрированной среды, либо ввести параметр при запуске компилятора, работающего в режиме командной строки.

В следующей таблице сведены различные модели и их сравнение друг с другом. Модели часто группируются по модели

кода или данных на малые (64K) и большие (16M); эти группы соответственно отражены в столбцах и строках таблицы.

Модели **tiny**, **small** и **compact** относятся к малым моделям кода, поскольку по умолчанию указатели кода являются ближними (**near**). Аналогичным образом, модели **compact**, **large huge** относятся к большим моделями данных, поскольку по умолчанию указатели на данные являются дальними (**far**).

Модели памяти

Размер данных	Размер кода	
	64K	16Мб
64K	Tiny (данные и код перекрываются; общий размер = 64K)	
	Small (без перекрытия; общий размер = 128K)	Medium (данные small, код large)
16Мб	Compact (данные large, код small)	Large (данные и код large)
		Huge (то же, что и large, но статические данные > 64K)

При компиляции модуля (некоторый исходный файл с несколькими подпрограммами), результирующий код для этого модуля не может превышать 64K, поскольку весь файл должен компилироваться в один кодовый сегмент. Это верно и в том случае, когда вы используете одну из больших моделей памяти (**medium**, **large** или **huge**). Если ваш модуль слишком велик и не помещается в одном кодовом сегменте (64K), вы должны разбить его на несколько файлов исходного кода, скомпилировать каждый из них по отдельности и затем скомпоновать их в одну программу. Аналогичным образом, хотя модель **huge** и позволяет иметь размер статических данных больше чем 64K, в каждом отдельном модуле статические данные не должны превышать 64K.

Программирование со смешанными моделями и модификаторы адресации

Borland C ++ вводит восемь новых ключевых слов, отсутствующих в языке Си стандарта ANSI (**near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_ss** и **_seg**), которые с некоторыми ограничениями и предупреждениями могут использоваться в качестве модификаторов для указателей (и в некоторых случаях, для функций).

В Borland C++ при помощи ключевых слов **near**, **far** или **huge** вы можете модифицировать объявления функций и указателей. Указатели данных **near**, **far** и **huge** рассматривались выше. Объекты **far** объявляются при помощи ключевого слова **far**. Функции **near** запускаются при помощи ближних вызовов (**near**), а выход из них происходит с использованием ближних команд возврата. Аналогичным образом, функции **far** вызываются дальними вызовами (**far**) и выполняют дальний (**far**) возврат. Функции **huge** похожи на функции **far**, за исключением того, что функции **huge** устанавливают регистр **DS** в новое значение, тогда как функции **far** не изменяют значения этого регистра.

Существует также четыре специальных ближних (**near**) указателя данных: **__cs**, **__ds**, **__es** и **__ss**. Имеются 16-битовые указатели, конкретно связанные с соответствующими сегментными регистрами. Например, если вы объявите указатель следующим образом:

```
char _ss *p;
```

то **p** будет содержать 16-битовое смещение в сегмент стека.

Функции и указатели в данной программе по умолчанию бывают ближними или дальними, в зависимости от выбранной модели памяти. Если функция или указатель являются ближними, то они автоматически связываются с регистром **CS** или **DS**.

В следующей таблице показано, как это происходит. Отметим, что размер указателя соответствует предельному размеру памяти, равному 64K (ближний, в пределах сегмента) или 1 Мб (дальний, содержит собственный адрес сегмента).

Типы указателей

Модель памяти	Указатели функции	Указатели данных
Tiny	near, _cs	near, _ds
Small	near, _cs	near, _ds
Medium	far	near, _ds
Compact	near, _cs	far
Large	far	far
Huge	far	far

Указатели сегментов

В объявлениях типа указателя сегмента используется **__seg**. В результате получаются 16-битовые указатели сегментов. Синтаксис **__seg** следующий:

```
тип_данных _seg *идентификатор
```

Например,

```
int _seg *name
```

Любое обращение по ссылке через «идентификатор» предполагает смещение 0. В арифметических операциях с указателями выполняются следующие правила:

- Нельзя использовать с указателями сегментов операции ++, --, +- или -=.
- Нельзя вычитать один указатель сегмента из другого.
- При сложении сегментного указателя с ближним (**near**) указателем результатом будет дальний (**far**) указатель, который формируется из сегмента, задаваемого сегментным указателем, и смещения из ближнего указателя. Эта операция разрешена только в том случае, если два указателя указывают на один и тот же тип, либо если один из указателей указывает на тип **void**. Независимо от указываемого типа умножение смещения не происходит.
- Когда сегментный указатель используется в выражении обращения по ссылке, он также неявно преобразуется в дальний указатель.
- При выполнении операции сложения или вычитания целочисленного операнда и сегментного указателя

результатом является дальний указатель, где сегмент берется из сегментного указателя, а смещение получается умножением размера объекта, на который указывает целочисленный операнд. Арифметическая операция выполняется таким образом, как если бы целое складывалось с указателем **far** или вычиталось из него.

- Сегментные указатели могут присваиваться, инициализироваться, передаваться в функции и из функций, сравниваться, и т.д. (Сегментные указатели сравниваются по правилам для **unsigned int**). Другими словами, за исключением перечисленных выше ограничений, они обрабатываются так же, как и любые другие указатели.

Объявление дальних объектов

Borland C++ позволяет объявлять дальние (**far**) объекты. Например:

```
int far x = 5;
int far z;
extern int far y = 4;
static long j;
```

Компилятор Borland C++ создает для каждого дальнего объекта отдельный сегмент. Параметры компилятора командной строки **-zE**, **-zF** и **-zH** (которые могут также задаваться директивой #pragma option) влияют на имя, класс и группу дальнего сегмента, соответственно. Изменяя эти значения при помощи указания **#pragma option**, вы тем самым распространяете новые установки на все объявления дальних объектов. Таким образом, для создания в конкретном сегменте дальнего объекта, можно использовать следующую последовательность:

```
#pragma option -zEmysegment -zHmygroup -zFmyclass
int far x;
#pragma option -zE* =zH* -zF*
```

Тем самым **x** будет помещен в сегмент **MYSEGMENT** с классом **MYCLASS** в группе **MYGROUP**, после чего все дальние объекты будут сброшены в значения, используемые по умолчанию. Отметим, что при использовании этих параметров можно поместить несколько дальних объектов в один сегмент:

```
#pragma option -zEcombined -zFmyclass
int far x;
```

```
double far y;
#pragma option -zE* -zF*
```

И **x**, и **y** окажутся в сегменте **COMBINED 'MYCLASS'**, без группы.

Объявление ближних или дальних функций

В некоторых случаях вам может потребоваться переопределить заданное по умолчание значение типа функции для модели памяти. Например, вы используете модель памяти **large**, и в программе имеется рекурсивная функция:

```
double  power(double x,int exp)
{
 if (exp <= 0)
    return(1);
 else
    return(x * power(x, exp-1));
}
```

Каждый раз, когда функция **power** вызывает сама себя, она должна выполнить дальний вызов, причем используется дополнительное пространства стека и число тактовых циклов. Объявив **power** как **near**, можно ускорить выполнение ее благодаря тому, что вызовы этой функции будут ближними:

```
double __near power(double x,int exp)
```

Это гарантирует, что функция **power** может вызываться только из того кодового сегмента, в котором она компилировалась, и что все обращения к ней будут ближними.

Это означает, что при использовании большой модели памяти (**medium**, **large** или **huge**) функцию **power** можно вызывать только из того модуля, в котором она определена. Прочие модули имеют свои собственные кодовые сегменты и не могут вызывать функции **near** из других модулей. Более того, ближняя функция до первого к ней обращения должна быть либо определена, либо объявлена, иначе компилятор не знает о необходимости генерировать ближний вызов.

И наоборот, объявление функции как дальней означает генерацию дальнего возврата. В малых моделях кодовой памяти дальняя функция должна быть объявлена или определена до первого к ней обращения, что обеспечит дальний вызов.

Вернемся к примеру функции **power**. Хорошо также объявить **power** как **static**, поскольку предусматривается вызывать ее только из текущего модуля. Если функция будет объявлена как **static**, то имя ее не будет доступно ни одной функции вне данного модуля.

Объявление указателей near, far или huge

Только что были рассмотрены случаи, в которых может понадобиться объявить функцию с другой моделью памяти, нежели остальная часть программы. Зачем то же самое может понадобиться для указателей? По тем же причинам, что и для функций: либо для улучшения характеристик быстродействия (объявив **__near** там, где по умолчанию было бы **__far**), либо для ссылки за пределы сегмента по умолчанию (объявив **__far** или **__huge** там, где по умолчанию бывает **__near**).

Разумеется, при объявлении функций или указателей с другим типом, нежели используемый по умолчанию, потенциально появляется возможность ошибок. Предположим, имеется следующий пример программы с моделью **small**:

```
void myputs(s)
char *s;
{
   int i;
   for (i = 0; s[i] != 0; i++) putc(s[i]);
}
main()
{
   char near *mystr;
   mystr = "Hello, world\n";
   myputs(mystr);
}
```

Эта программа работает удовлетворительно, хотя объявление **mystr** как **__near** избыточно, поскольку все указатели, как кода, так и данных, будут ближними (**near**) по умолчанию.

Однако, что произойдет, если перекомпилировать эту программу с моделью памяти **compact** (либо **large** или **huge**)? Указатель **mystr** в функции **main** останется ближним (16-битовым). Однако, указатель **s** в функции **myputs** теперь будет дальним (**far**), поскольку по умолчанию теперь используется **far**. Это означает, что попытка создания дальнего указателя приведет

к извлечению из стека двух слов, и полученный таким образом адрес, безусловно, не будет являться адресом функции **mystr**.

Как избежать этой проблемы? Решение состоит в том, чтобы определить **myputs** в современном стиле Си:

```
void myputs(char *s)
{
   /* тело myputs */
}
```

Теперь при компиляции вашей программы Borland C++ знает, что **myputs** ожидает указатель на **char**. Поскольку компиляция выполняется с моделью **large**, то известно, что указатель должен быть **__far**. Вследствие этого Borland C++ поместит в стек регистр сегмента данных (**DS**) и 16-битовое значение **mystr**, образуя тем самым дальний указатель.

Если вы собираетесь явно объявлять указатели как **far** или **near**, не забывайте использовать прототипы тех функций, которые могут работать с этими указателями.

Как быть в обратном случае: когда аргументы **myputs** объявлены как **__far**, а компиляция выполняется с моделью памяти **small**? И в этом случае без прототипа функции у вас возникнут проблемы, поскольку функция **main** будет помещать в стек и смещение, и адрес сегмента, тогда как **myputs** будет ожидать приема только одного смещения. При наличии определений функций в прототипах **main** будет помещать в стек только смещение.

Создание указателя данного адреса «сегмент:смещение»

Как создать дальний указатель на конкретный адрес памяти (конкретный адрес «**сегмент:смещение**»)? Для этого можно воспользоваться встроенной библиотечной подпрограммой **MK_FP**, которая в качестве аргумента воспринимает сегмент и смещение, и возвращает дальний указатель. Например:

```
MK_FP(segment_value, offset_value)
```

Имея дальний указатель **fp**, вы можете получить значение сегмента полного адреса с помощью **FP_SEG(fp)** и значение смещения с помощью **FP_OFF(fp)**.

Использование библиотечных файлов

Borland C++ предлагает для каждой из шести моделей памяти собственную версию библиотеки стандартных подпрограмм. Компилятор Borland C++ при этом проявляет достаточно «интеллекта», чтобы при последующей компоновке брать нужные библиотеки и в нужной последовательности, в зависимости от выбранной вами модели памяти. Однако, при непосредственном использовании компоновщика Borland C++ **TLINK** (как автономного компоновщика) вы должны явно указывать используемые библиотеки.

Компоновка смешанных модулей

Что произойдет, если вы компилируете один модуль с использованием модели памяти **small** (малая), второй — модели **large** (большая), и затем хотите скомпоновать их? Что при этом произойдет?

Файлы скомпонуются удовлетворительно, но при этом вы столкнетесь с проблемами. Если функция модуля с моделью **small** вызывает функцию в модуле с моделью **large**, она будет использовать при этом ближний вызов, что даст абсолютно неверные результаты. Кроме того, у вас возникнут проблемы с указателями, поскольку функция в модуле **small** ожидает, что принимаемые и передаваемые ей указатели будут **__near**, тогда как функция в модуле **large** ожидает работу с указателями **__far**.

И снова решение заключается в использовании прототипов функций. Предположим, что вы поместили **myputs** в отдельный модуль и скомпилировали его с моделью памяти **large**. Затем вы создаете файл заголовка **myputs.h** (либо с любым другим именем и расширением **.h**), который содержит следующий прототип функции:

```
void far myputs(char far *s);
```

Теперь, если поместить функцию **main** в отдельный модуль (**MYMAIN.C**) и выполнить следующие установки:

```
#include <stdio.h>
#include "myputs.h"
main()
{
   char near *mystr;
   mystr = "Hello, world\n";
   myputs(mystr);
}
```

то при компиляции данной программы Borland C++ считает прототип функции из файла **MYPUTS.H** и увидит, что это функция **__far**, ожидающая указатель **__far**. В результате этого даже при модели памяти **small** при компиляции будет сгенерирован правильный вызывающий код.

Как быть, если помимо этого вам требуется компоновка с библиотечными подпрограммами? Лучший подход здесь заключается в том, чтобы выбрать одну из библиотек с моделью **large** и объявить все как **far**. Для этого сделайте копии всех файлов заголовка, которые вы обычно включаете (таких, как **stdio.h**) и переименуйте эти копии (например, **fstdio.h**).

Затем отредактируйте копии прототипов функций таким образом, чтобы там было явно указано **far**, например:

```
int far cdecl printf(char far* format, ...);
```

Тем самым, не только вызовы подпрограмм будут дальними, но и передаваемые указатели также будут дальними. Модифицируйте вашу программу таким образом, чтобы она включала новый файл заголовка:

```
#include <fstdio.h>
main()
{
    char near *mystr;
    mystr = "Hello, world\n";
    printf(mystr);
}
```

Скомпилируйте вашу программу при помощи компилятора **BCC**, затем скомпонуйте ее при помощью утилиты **TLINK**, указав библиотеки с моделью памяти **large**, например **CL.LIB**. Смешивание модулей с разными моделями — вещь экстравагантная, но допустимая. Будьте, однако, готовы к тому, что любые неточности здесь приводят к ошибкам, которые очень трудно найти и исправить при отладке.

Оверлеи (VROOMM)

Оверлеями называются части кода программы, совместно использующие общую область памяти. Одновременно в памяти находятся только те части программы, которые в текущий момент нужны данной функции.

Оверлеи могут существенно снизить во время выполнения программы требования к выделяемой программе памяти. При помощи оверлеев можно создавать программы, значительно превышающие по размеру общую доступную память системы, поскольку одновременно в памяти находится лишь часть данной программы.

Оверлеи используются только в 16-разрядных программах DOS. В приложениях Windows для сокращения объема используемой памяти вы можете пометить сегменты как **Discardable** (выгружаемые).

Работа программ с оверлеями

Программа управления оверлеями (**VROOMM**, или Virtual Run-time Object-Oriented Memory Manager) выполняет за вас большую часть работы по организации оверлеев. В обычных оверлейных системах модули группируются в базовый и набор оверлейных модулей. Подпрограммы в данном оверлейном модуле могут вызывать подпрограммы из этого же модуля и из базового модуля, но не из других модулей. Оверлейные модули перекрывают друг друга, т.е. одновременно в памяти может находиться только один оверлейный модуль, и все они при активизации занимают один и тот же участок физической памяти. Общий объем памяти, необходимой для запуска данной программы, определяется размером базового, плюс максимального оверлейного модуля. Эта обычная схема не обеспечивает достаточной гибкости. Она требует полного учета всех возможных обращений между модулями программы и, соответственно, планируемой вами, группировки оверлеев. Если вы не можете разбить вашу программу в соответствии со взаимозависимостью обращений между ее модулями, то вы не сможете и разбить ее на оверлеи.

Схема VROOMM совершенно иная. Она обеспечивает динамический свопинг сегментов. Основной единицей свопинга является сегмент. Сегмент может состоять из одного или нескольких модулей. И что еще более важно, любой сегмент может вызывать любой другой сегмент. Вся память делится на базовую область и область свопинга. Как только встречается вызов функции, которая не находится ни в базовой, ни в области свопинга, сегмент, содержащий вызываемую функцию, помещается в область свопинга, возможно, выгружая оттуда при

этом другие сегменты. Это мощное средство — подобное виртуальной программной памяти. От вас больше не требуется разбивать код на статические, отдельные оверлейные блоки. Вы просто запускаете программу!

Что происходит, когда возникает необходимость поместить сегмент в область свопинга? Если эта область имеет достаточно свободного места, то данная задача выполняется просто. Если же нет, то из области свопинга, чтобы искомая свободная область освободилась, должен быть выгружен один или более сегментов. Как выбрать сегменты для выгрузки? Действующий здесь алгоритм очень сложен. Упрощенная версия его такова: если в области свопинга имеется неактивный сегмент, то для выгрузки выбирается он. Неактивными считаются сегменты, в которых в текущий момент нет выполняемых функций. В противном случае берется активный сегмент. Удаление сегментов из памяти продолжается до тех пор, пока в области свопинга не образуется достаточно свободной памяти для размещения там требуемого сегмента. Такой метод называется динамическим свопингом.

Чем больше памяти выделено для области свопинга, тем лучше работает программа. Область свопинга работает как кэш-память: чем больше кэш, тем быстрее работает программа. Наилучшие значения размера области свопинга определяются размерами рабочего множества данной программы.

После загрузки оверлея в память он помещается в оверлейный буфер, который расположен в памяти между сегментом стека и дальней динамически распределяемой областью. По умолчанию размер оверлейного буфера вычисляется и устанавливается при загрузке программы, но его можно изменить при помощи глобальной переменной **_ovrbuffer**. Если достаточный размер памяти недоступен, то появляется либо сообщение об ошибке DOS («Program too big to fit in memory» — «Программа слишком велика для имеющейся памяти»).

Важной возможностью программы управления оверлеями является ее способность при удалении моделей из оверлейного буфера выполнять их свопинг с дополнительной расширенной памятью. Следующий раз, как только данный модуль понадобится, он в этом случае будет не считываться с диска, а просто копироваться из этой памяти. Это существенно ускоряет свопинг.

При использовании оверлеев память распределяется, как показано на следующем рисунке:

Распределение памяти для оверлейных структур

```
                   Модель MEDIUM                       Модель LARGE

                   класс CODE       Резидентный    класс CODE
                                    код

Эти сегменты       класс OVRINFO    Данные для     класс OVRINFO
генерируются                        управления
компоновщиком                       оверлеями
автоматически

                   класс STUBSEG    Один сегмент   класс STUBSEG
                                    stub для
                                    каждого
                                    оверлейного
                                    сегмента

                          _DATA                           _DATA
Ближняя динами-      класс DATA                      класс DATA
чески распреде-
ляемая область и     ближняя куча      Отдельный
стек совместно     ^                   сегмент     ^
используют сег-           стек         стека              стек
мент данных
                  оверлейный буфер               оверлейный буфер
                  (распределяется                (распределяется
                   при загрузке)                  при загрузке)

                       дальняя                        дальняя
                     динамически                    динамически
                    распределяемая                 распределяемая
                   v    область                    v    область

                                               Модель HUGE

                                  Резидентный    класс CODE
                                  код

             Эти сегменты         Данные для     класс OVRINFO
             генерируются         управления
             компоновщиком        оверлеями
             автоматически

                                  Один дополни-  класс STUBSEG
                                  тельный сег-
                                  мент для
                                  каждого
                                  оверлейного
                                  сегмента

   . . .                          Несколько
                                  сегментов
                                  данных

                                  Отдельный
                                  сегмент      ^
                                  стека               стек

                                             оверлейный буфер
                                             (распределяется
                                              при загрузке)

                                                  дальняя
                                                динамически
                                               распределяемая
                                             v    область
```

Оптимальное использования оверлеев Borland C++

Для полного использования преимуществ оверлейных структур, создаваемых Borland C++, сделайте следующее:

- Минимизируйте резидентный код (резидентные библиотеки исполняющей системы, обработчики прерываний и драйверы устройств).
- Установите размер оверлейного пула таким образом, чтобы добиться наиболее комфортных условий для создаваемой программы (начните со 128K и регулируйте этот размер вверх и вниз, пока не установите желаемое соотношение между быстродействием и размером программы).
- Подумайте об универсальности и многосторонности создаваемого кода: воспользуйтесь преимуществами оверлейной структуры и обеспечьте поддержку обработки специальных случаев, интерактивную справочную систему по программе и прочие не рассматриваемые здесь средства, являющиеся достоинствами с точки зрения конечного пользователя.

Требования

При создании оверлеев следует помнить несколько простых правил, а именно:

- Минимальная часть программы, которая может выделяться в качестве оверлея, это сегмент.
- Прикладные программы с оверлейной структурой должны иметь одну из трех следующих моделей памяти: **medium**, **large** или **huge**; модели **tiny**, **small** и **compact** оверлеи не поддерживают.
- Перекрывающиеся сегменты подчиняются обычным правилам слияния сегментов. То есть, в одном и том же сегменте может участвовать несколько объектных файлов.

Генерация оверлеев во время компоновки полностью не зависит от управления сегментами во время исполнения программы; компоновщик не включает автоматически каких-либо кодов для управления оверлеями. Действительно, с точки зрения компоновщика программа управления оверлеями является просто одним из подлежащих компоновке участков кода.

Единственное предположение, которое делает компоновщик, состоит в том, что программа управления оверлеями воспринимает вектор прерываний (обычно **INT 3FH**), через который происходит управление динамической загрузкой. Такой уровень «прозрачности» упрощает создание пользовательских программ управления оверлеями, наилучшим образом управляющих требованиям конкретной прикладной программы.

Оверлеи и обработка исключительных ситуаций

Если вы пишете оверлейную программу, содержащую конструкции для обработки исключительных ситуаций, то существует ряд ситуаций, которых следует избегать. Следующие программные элементы не могут содержать конструкцию обработки исключительных ситуаций:

- Не расширяемые подставляемые функции.
- Шаблоны функций.
- Функции-элементы или шаблоны классов.

Конструкция обработки исключительной ситуации включает в себя написанный пользователем блок **try/catch** и **__try/__except**. Кроме того, компилятор также может включать обработчики исключительных ситуаций и блоки с локальными динамическими переменными, спецификациями исключительных ситуаций и некоторые выражения **new/delete**.

Если вы пытаетесь использовать в оверлее вышеуказанные конструкции обработки исключительных ситуаций, компоновщик идентифицирует функцию и модуль следующим сообщением:

```
Error: Illegal local public in функция in module модуль
```

Когда эта ошибка вызывается подставляемой функцией, вы можете переписать функцию таким образом, чтобы она не была подставляемой. Если это вызвано шаблоном функции, можно сделать следующее:

- удалить из функции все конструкции обработки исключительной ситуации;
- удалить функцию их оверлейного модуля.

Особенно внимательно нужно строить оверлейную программу, которая использует множественное наследование. Попытка создать оверлейный модуль, который определяет или

использует конструкторы или деструкторы классов с множественным наследованием может привести к тому, что компоновщик будет генерировать следующее сообщение об ошибке:

```
Error: Illegal local public in класс: in module модуль
```

Когда генерируется такое сообщение, идентифицированный компоновщиком модуль не следует делать оверлейным.

В классе контейнера (в **BIDS?.LIB**) есть механизм обработки исключительной ситуации, который по умолчанию выключен. Однако диагностическая версия генерирует исключительные ситуации и не может использоваться в оверлеях. По умолчанию класс string может генерировать исключительные ситуации, и его не следует использовать в программах с оверлеями.

Использование оверлеев

Для создания программы с оверлейной структурой все ее модули должны компилироваться с включенным параметром компилятора **-Y**. Для того, чтобы сделать оверлейным конкретный модуль, его следует компилировать с параметром **-Yo**. (**-Yo** автоматически включает параметр **-Y**).

Параметр **-Yo** распространяется на все модули и библиотеки, следующие за ней в командной строке компилятора **BCC**. Отменить ее можно, задав **-Yo-**. Эти два параметра являются единственными параметрами командной строки, которые могут следовать после имен файлов. Например, для того, чтобы сделать оверлейным модуль **OVL.C**, но не библиотеку **GRAPHICS.LIB**, можно использовать любую из следующих командных строк:

```
BCC -ml - Yo ovl.c -Yo- graphics.lib
```

или

```
BCC -ml graphics.lib -Yo ovl.c
```

Если при запуске компоновщика **TLINK** явно задана компоновка файла **.EXE**, то в командной строке компоновщика должен задаваться параметр **/o**.

Предположим, вы хотите иметь оверлейную структуру в программе, состоящей из трех модулей: **MAIN.C**, **O1.C** и **O2.C**. Оверлеями должны являться модули **O1.C** и **O2.C**. (Программа **MAIN.C** содержит зависящие от текущего времени подпрограммы и обработчики прерываний и потому должна оставаться резидентной). Предположим, что данная программа использует

модель памяти **large**.

Следующая команда позволяет выполнить данную задачу:

```
BCC -ml -Y main.c -Yo o1.c o2.c
```

В результате получится выполняемый файл **MAIN.EXE** с двумя оверлеями.

Разработка программ с оверлеями

Этот раздел содержит важные сведения о разработке программ с оверлеями с хорошими характеристиками.

При компиляции оверлейного модуля вы должны использовать большую модель памяти (**medium**, **large** или **huge**). При всяком вызове функции из оверлейного модуля вы обязаны гарантировать, что все активные в текущий момент функции являются дальними.

Вы обязаны компилировать все оверлейные модули с параметром **-Y**, что обеспечит оверлейную структуру генерируемого кода.

Невыполнение требования дальних вызовов в оверлейной программе приведет при выполнении программы к непредсказуемым и возможно, катастрофическим результатам.

Размер оверлейного буфера по умолчанию в два раза превышает размер самого большого оверлея. Для большинства прикладных программ такое умолчание вполне адекватно. Однако, представим себе ситуацию, когда какая-либо функция программы реализована несколькими модулями, каждый из которых является оверлейным. Если общий размер этих модулей превышает размер оверлейного буфера, то если модули часто вызывают друг друга, это приведет к интенсивному свопингу.

Очевидно, что решение здесь заключается в увеличении размера оверлейного буфера до таких размеров, чтобы в любой момент времени в нем помещались все часто вызывающие друг друга оверлеи. Это можно сделать, установив через глобальную переменную **_ovrbuffer** требуемый размер в параграфах. Например, для установки размера оверлейного буфера равным 128K, включите в ваш код следующий оператор:

```
unsigned _ovrbuffer = 0x2000;
```

Общей формулы для определения идеального размера

оверлейного буфера не существует.

Не создавайте оверлейных модулей, содержащих обработчики прерываний, а также в случаях небольших или критичных к быстродействию программ. Вследствие нереентерабельной природы операционной системы DOS модули, которые могут вызываться функциями прерываний, не должны быть оверлейными.

Программа управления оверлеями Borland C++ полностью поддерживает передачу оверлейных функций как аргументов, присвоение и инициализацию переменных типа указателя функции, адресующих оверлейные функции, а также вызов оверлейных подпрограмм через указатели функций.

Отладка оверлейных программ

Большинство отладчиков либо имеет весьма ограниченные средства отладки программ с оверлейной структурой, либо вообще не имеет таких средств. Иначе дело обстоит с интегрированным со средой разработки программ отладчиком Borland C++ и автономным отладчиком фирмы Borland (Turbo Debugger). Оба эти отладчика полностью поддерживают пошаговую отладку и установку точек останова в оверлеях совершенно «прозрачным» для вас способом. Благодаря использованию оверлеев вы имеете возможность легко разрабатывать и отлаживать громоздкие прикладные программы — как в интегрированной среде, так и при помощи Turbo Debugger.

Внешние подпрограммы в оверлеях

Подобно обычным функциям языка Си, внешние (external) подпрограммы на языке Ассемблера, чтобы хорошо работать с подсистемой управления оверлеями, должны подчиняться некоторым правилам.

Если подпрограмма на языке ассемблера выполняет вызов любой оверлейной функции, то такая подпрограмма должна иметь объявление **FAR** и устанавливать границу стека при помощи регистра **BP**. Например, если **OtherFunc** — это оверлейная функция в другом модуле, и ее вызывает подпрограмма на языке Ассемблера **ExternFunc**, то тогда

ExternFunc должна быть дальней (**FAR**) и устанавливать границы

стека, как показано ниже:

```
ExternFunc      PROC        FAR
   push         bp                ; сохранить bp
   mov          bp,sp             ; установить стек
   sub          sp,LocalSize      ; распределить
                                  ; локальные
                                  ; переменные
...
   call         OtherFunc         ; вызов другого
                                  ; оверлейного
                                  ; модуля
...
   mov          sp,bp             ; освобождение
                                  ; локальных
                                  ; переменных
   pop          bp                ; восстановление BP
   RET                            ; возврат
ExternFunc      ENDP
```

где **LocalSize** — это размер локальных переменных. Если **LocalSize** равен нулю, вы можете опустить две строки распределения и освобождения локальных переменных, но ни в коем случае нельзя опускать установку границ стека **BP**, даже если аргументов и переменных в стеке нет.

Эти требования остаются теми же в случае, когда **ExternFunc** делает косвенные ссылки на оверлейные функции. Например, если **OtherFunc** вызывает оверлейные функции, но сама не является оверлейной, то **ExternFunc** должна быть **FAR** и также должна устанавливать границы стека.

В случае, когда ассемблерная подпрограмма не делает ни прямых, ни косвенных ссылок на оверлейные функции, то специальные требования отсутствуют; подпрограмма на языке Ассемблера может быть объявлена как **NEAR**. Она не обязана устанавливать границ стека.

Оверлейные подпрограммы на языке ассемблера не должны создавать переменные в кодовом сегменте, поскольку все изменения, внесенные в оверлейный кодовый сегмент, теряются при освобождении оверлея. Подобным же образом, указатели объектов, расположенных в оверлейных сегментах, не сохраняют достоверность после вызова других оверлеев, поскольку

программа управления оверлеями свободно перемещает и освобождает оверлейные кодовые сегменты в памяти.

Свопинг

Если в системе компьютера установлена дополнительная или расширенная память, вы можете сообщить программе управления оверлеев, что он должен использовать эту память при свопинге. В этом случае при удалении модуля из оверлейного буфера (когда туда требуется загрузить новый модуль, а буфер полон) программа управления оверлеями может поместить удаляемый модуль в эту память. При любой последующей загрузке этого модуля за счет того, что модуль перемещается в памяти, а не считывается с диска, экономится время.

В обоих случаях есть две возможности: программа управления оверлеями может либо обнаруживать наличие дополнительной или расширенной памяти самостоятельно и затем брать на себя управление этой памятью, либо использовать уже обнаруженную и распределенную часть такой памяти. В случае расширенной памяти обнаружение памяти не во всех случаях выполняется удачно, поскольку многие программы кэширования памяти и программы организации виртуального диска могут использовать эту память, не делая об этом никаких отметок. Чтобы избежать этих проблем, вы должны сообщить программе управления оверлеями начальный адрес расширенной памяти и какой участок ее можно безопасно использовать. Borland C++ предусматривает две функции, которые позволяют вам инициализировать расширенную и дополнительную память — **_OvrInitEms** и **_OvrInitExt**.

Математические операции

Ниже рассматриваются возможности работы с числами с плавающей точкой и объясняется, как использовать математические операции с комплексными числами.

Операции ввода-вывода с плавающей точкой

Ввод вывод с плавающей точкой требует компоновки с подпрограммами преобразования, используемыми функциями **printf** и **scanf**. Чтобы уменьшить размер выполняемого файла, форматы с плавающей точкой автоматически не компонуются.

Однако такая компоновка автоматически выполняется при использовании в программе математической подпрограммы или получении адреса некоторого числа с плавающей точкой. Если не выполняется ни одно из этих действий не выполняется, то отсутствие форматов с плавающей точкой может дать в результате ошибку ввода-вывода. Правильно оформить программу можно, например, следующим образом:

```
/* Подготовка к выводу чисел с плавающей точкой */
#include <stdio.h>
#pragma extref _floatconvert
void main()
{ printf(*d = %f\n", 1.3);
}
```

Сопроцессор

Си работает с двумя числовыми типами: целыми (**int**, **short**, **long** и т.д.) и с плавающей точкой (**float double** и **long double**). Процессор вашего компьютера легко справляется с обработкой чисел целых типов, однако числа с плавающей точкой отнимают больше времени и усилий.

Семейство процессоров iAPx86 имеет сопутствующее ему семейство математических сопроцессоров. Мы будем обозначать все семейство математических сопроцессоров термином «сопроцессор». (В случае процессора 80487 вы имеете математический сопроцессор уже встроенным в основной.)

Процессор 80x87 представляет собой специальный аппаратно реализованный числовой процессор, который можно установить на вашем PC. Он служит для выполнения с большой скоростью команд с плавающей точкой. При большом количестве в вашей программе операций с плавающей точкой вам, безусловно, нужен сопроцессор. Блок центрального процессора в вашем компьютере осуществляет интерфейс с 80x87 по специальным шинам интерфейса.

Эмуляция платы 80x87

По умолчанию в Borland C++ устанавливается параметр генерации кода «эмуляция» (параметр компилятора режима командной строки **-f**). Этот параметр предназначен для программ, которые могут вообще не иметь операций с плавающей точкой, а также для программ, которые должны выполняться и на

машинах, на которых сопроцессор 80x87 не установлен.

В случае параметра эмуляции компилятор генерирует код, как если бы сопроцессор присутствовал, но при компоновке подключает библиотеку эмуляции операций с плавающей точкой (**EMU.LIB**). При выполнении такой программы сопроцессор 80x87, если он установлен, будет использоваться. Если же во время выполнения процессора не окажется, то программа будет использовать специальное программное обеспечение, эмулирующее процессор 80x87.

Использование кода 80x87

Если вы планируете использовать вашу программу исключительно на машинах с установленным математическим сопроцессором 80x87, то можно сэкономить около 10К памяти программы, опустив из нее логику автоматического определения присутствия процессора 80x87 и эмулятора. Для этого следует просто выбрать параметр генерации кода операций с плавающей точкой при наличии сопроцессора 80x87 (или параметр компилятора режима командной строки **-f87**). Borland C++ в этом случае скомпонует вашу программу с библиотекой **FP87.LIB** (вместо **EMU.LIB**).

Получение кода без операций с плавающей точкой

При отсутствии в программе операций с плавающей точкой вы можете сэкономить немного времени компиляции, выбрав значение параметра генерации операций с плавающей точкой None («отсутствуют») (или параметр компилятора командной строки **-f-**). Тогда Borland C++ не будет выполнять компоновку ни с библиотекой **EMU.LIB**, ни с **FP87.LIB**, ни с **MATHx.LIB**.

Параметр быстрых вычислений с плавающей точкой

Borland C++ имеет параметр быстрых вычислений с плавающей точкой (параметр компилятора режима командной строки **-ff**). Выключить этот параметр можно при помощи параметра командной строки **-ff-**. Его назначение состоит в выполнении некоторой оптимизации, противоречащей правильной семантике языка Си. Например:

```
double x;
x = (float)(3.5*x);
```

Для вычисления по обычным правилам **x** умножается на 3.5,

давая точность результата **double**, которая затем усекается до точности **float**, после чего **x** записывается как **double**. При использовании параметра быстрых вычислений с плавающей точкой произведение типа **long double** преобразуется непосредственно в **double**. Поскольку лишь очень немногие программы чувствительны к потере точности при преобразовании от более точного к менее точному типу с плавающей точкой, то данный параметр используется по умолчанию.

Переменная операционной среды 87

При построении программы с эмуляцией сопроцессора 80x87, которая устанавливается по умолчанию, ваша программа станет автоматически проверять наличие сопроцессора 80x87 и использовать его, если он установлен в машине.

Существует ряд ситуаций, в которых вам может понадобиться отменить режим автоматического определения наличия сопроцессора по умолчанию. Например, ваша собственная исполняющая система может иметь сопроцессор 80x87, но вам требуется проверить, будет ли программа работать так, как вы предполагали, в системе без сопроцессора. Либо ваша программа предназначена для работы в системе, совместимой с PC, но данная конкретная система возвращает логике автоматического определения наличия сопроцессора неверную информацию (либо при отсутствии сопроцессора 80x87 говорит, что он на месте, либо наоборот).

Borland C++ имеет параметр для переопределения логики определения наличия сопроцессора при загрузке программы. Этот параметр — соответствующая переменная операционной среды системы 87. Переменная операционной среды 87 устанавливается в ответ на подсказку DOS при помощи команды **SET**:

```
C>SET 87=N
```

или

```
C>SET 87=Y
```

Ни с какой стороны знака равенства не должно быть пробелов. Установка переменной операционной среды 87 в **N** говорит загрузочному коду исполняющей системы о том, что вы

не хотите использовать сопроцессор 80x87 даже в том случае, если

он установлен в системе.

Установка переменной операционной среды в значение **Y** означает, что сопроцессор на месте и вы желаете, чтобы программа его использовала. Программист должен знать следующее: если установить **87=Y**, а физически сопроцессор 80x87 в системе не установлен, то система «зависнет».

Если переменная операционной среды 87 была определена (с любым значением), и вы желаете сделать ее неопределенной, введите в ответ на подсказку DOS:

```
C>SET=
```

Непосредственно после знака равенства нажмите клавишу **Enter**, и переменная 87 станет неопределенной.

Регистры и сопроцессор 80x87

При работе с плавающей точкой вы должны учитывать два момента, связанных с использованием регистров:

- В режиме эмуляции сопроцессора 80x87 циклический переход в регистрах, а также ряд других особенностей 80x87 не поддерживается.
- Если вы смешиваете операции с плавающей точкой и встроенные коды на языке Ассемблера, то при использовании регистров следует должны принимать некоторые меры предосторожности. Это связано с тем, что набор регистров сопроцессора 80x87 перед вызовом функции в Borland C++ очищается. Вам может понадобиться извлечь из стека и сохранить регистры сопроцессора 80x87 до вызова функции, использующей сопроцессор, если вы не уверены, что свободных регистров достаточно.

Отмена обработки особых ситуаций для операций с плавающей точкой

По умолчанию программа Borland C++ в случае переполнения или деления на ноль в операциях с плавающей точкой аварийно прерывается. Вы можете замаскировать эти особые ситуации для операций с плавающей точкой, вызывая в **main _control87** перед любой операцией с плавающей точкой.

Например:

```
#include <floar.h>
main() {
_control87(MCW_EM,MCW_EM);
...
}
```

Можно определить особую ситуацию для операции с плавающей точкой, вызвав функции **_status87** или **_clear87**.

Определенные математические ошибки могут также произойти в библиотечных функциях, например, при попытке извлечения квадратного корня из отрицательного числа. По умолчанию в таких случаях выполняется вывод на экран сообщений об ошибке и возврат значения **NAN** (код IEEE «not-a-number» — «не число»). Использование **NAN** (нечисловых значений) скорее всего приведет далее к возникновению особой ситуации с плавающей точкой, которая в свою очередь вызовет, если она не замаскирована, аварийное прерывание программы. Если вы не желаете, чтобы сообщение выводилось на экран, вставьте в программу соответствующую версию **matherr**.

```
#include <math.h>
int cdecl matherr(struct exception *e)
{
return 1;       /* ошибка обработана */
}
```

Любое другое использование **matherr** для внутренней обработки математических ошибок недопустимо, так как она считается устаревшей и может не поддерживаться последующими версиями Borland C++.

Математические операции с комплексными числами

Комплексными называются числа вида **x + yi**, где **x** и **y** — это вещественные числа, а **i** — это корень квадратный из **-1**. В Borland C++ всегда существовал тип:

```
struct complex
{
double x, y;
};
```

определенный в **math.h**. Этот тип удобен для представления комплексных чисел, поскольку их можно рассматривать в

качестве пары вещественных чисел. Однако, ограничения Си делают арифметические операции с комплексными числами несколько громоздкими. В C++ операции с комплексными числами выполняются несколько проще.

Для работы с комплексными числами в C++ достаточно включить файл **complex.h**. В **complex.h** для обработки комплексных чисел переопределены:

- все обычные арифметические операции;
- операции потоков >> и <<;
- обычные арифметические функции, такие как **sqrt** и **log**.

Библиотека **complex** активизируется только при наличии аргументов типа **complex**. Таким образом, для получении комплексного квадратного корня из **-1** используйте:

```
sqrt(complex(-1))
```

а не

```
sqrt(-1)
```

Использование двоично-десятичной арифметики (BCD)

Borland C++, также как и большинство прочих компьютеров и компиляторов, выполняет математические вычисления с числами в двоичном представлении (то есть в системе счисления с основанием 2). Это иногда путает людей, привыкших исключительно к десятичной математике (в системе счисления с основанием 10). Многие числа с точным представлением в десятичной системе счисления, такие как 0.01, в двоичной системе счисления могут иметь лишь приближенные представления.

В большинстве прикладных программ двоичные числа предпочтительны, однако в некоторых ситуациях ошибка округления в преобразованиях между системами счисления с основаниями 2 и 10 нежелательна. Наиболее характерным случаем здесь являются финансовые или учетные задачи, где предполагается сложение центов. Рассмотрим программу, складывающую до 100 центов и вычитающую доллар:

```
#include <stdio.h>
int i;
float x =0.0;
for (i = 0; i < 100; ++i)
```

```
x += 0.01;
x -= 1.0;
print("100*.01 - 1 = %g\1",x);
```

Правильным ответом является 0.0, однако ответ, полученный данной программой, будет малой величиной, близкой к 0.0. При вычислении ошибка округления, возникающая во время преобразования 0.01 в двоичное число, накапливается. Изменение типа **x** на **double** или **long double** только уменьшает ошибку вычисления, но не устраняет ее вообще.

Для решения этой проблемы Borland C++ предлагает специфический для C++ тип **bcd** (двоично-десятичный), объявленный в файле **bcd.h**. В случае двоично-десятичного представления число 0.01 будет иметь точное значение, а переменная **x** типа **bcd** даст точное исчисление центов.

```
#include <bcd.h>
int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
x += 0.01;
x -= 1.0;
cout << "100*0.1 - 1 = " << x << "\n";
```

При этом необходимо учитывать следующие особенности типа **bcd**:

- **bcd** не уничтожает ошибку округления вообще. Вычисление типа 1.0/3.0 все равно будет иметь ошибку округления.
- Обычные математические функции, такие как **sqrt** и **log**, для аргументов с типом **bcd** переопределяются.
- Числа типа **bcd** имеют точность представления около 17 разрядов и диапазон принимаемых значений от $1x10^{-125}$ до $1x10^{125}$.

Преобразования двоично-десятичных чисел

Тип **bcd** — это определяемый тип, отличный от **float**, **double** или **long double**. Десятичная арифметика выполняется только когда хотя бы один операнд имеет тип **bcd**.

Для преобразования двоично-десятичного числа обратно к обычной системе счисления с основанием 2 (тип **float**, **double** или **long double**), служит функция-элемент **real** класса **bcd**. Функция

real выполняет все необходимые преобразования к типам **long**, **double** или **lognd double**, хотя такое преобразование не выполняется автоматически. Функция **real** выполняет все необходимые преобразования к типу **long double**, который может быть затем преобразован к другим типам при помощи обычных средств языка Си. Например:

```
/* Печать чисел BCD */
#include <bcd.h>
#include <iostream.h>
#include <stdio.h>
void main(void) {
     bcd a = 12.1;
     double x = real(a);    /* преобразование для печати
     printf("\na = %Lg", real(a));
     printf("\na = %g", (double)real(a));
     cout << "\na =" << a; /* рекомендуемый метод
```

Отметим, что поскольку **printf** не выполняет контроль типа аргументов, спецификатор формата должен иметь **L**, если передается значение **real(a)** типа **long double**.

Число десятичных знаков

Вы можете задать, сколько десятичных знаков должно участвовать в преобразовании из двоичного типа в **bcd**. Это число является вторым, необязательным аргументом в конструкторе **bcd**. Например, для преобразования $1000.00/7 в переменную **bcd**, округленную до ближайшего цента, можно записать:

```
bcd a = bcd(1000.00/7, 2)
```

где 2 обозначает два разряда после десятичной точки. Таким образом:

```
1000.00/7    =    142.85714
bcd(1000.00/7, 2)    =    142.860
bcd(1000.00/7, 1)    =    142.900
bcd(1000.00/7, 0)    =    142.000
bcd(1000.00/7, -1)    =    140.000
bcd(1000.00/7, -2)    =    100.000
```

Округление происходит по банковским правилам, что означает округление до ближайшего целого числа, причем в случае одинакового «расстояния» до ближайшего целого в прямую и обратную сторону округление выполняется в сторону четного

Например:

```
bcd(12.335, 2)   =   12.34
bcd(12.245, 2)   =   12.34
bcd(12.355, 2)   =   12.36
```

Такой метод округления задается стандартом IEEE.

Видео-функции

Borland C++ поставляется с полной библиотекой графических функций, позволяющих создание экранных графиков и диаграмм. Графические функции доступных только для 16-разрядных приложений DOS. Ниже приведено краткое описание видеорежимов и окон. Затем объясняется, как программировать в текстовом и графическом режимах.

Видеорежимы

Ваш компьютер обязательно имеет некоторый видеоадаптер. Это может быть монохромный дисплейный адаптер (MDA) для базового (только текстового) дисплея, либо это может быть графический адаптер, например цветной графический адаптер (CGA), улучшенный графический адаптер (EGA), монохромный графический адаптер Hercules или видеографическая матрица (VGA/SVGA). Каждый из этих адаптеров может работать в нескольких режимах. Режим определяет величину экрана — 80 или 40 символов в строке (только в текстовом режиме), разрешающую способность экрана (только в графическом режиме) и тип дисплея (цветной или черно-белый).

Рабочий режим экрана определяется, когда ваша программа вызывает одну из функций определения режима (**textmode**, **initgraph** или **setgraphmode**).

- В текстовом режиме экран компьютера разделен на ячейки (80 или 40 столбцов в ширину и 25, 43 или 50 строк по высоте). Каждая ячейка состоит из атрибута и символа. Символ представляет собой имеющий графическое отображение символ кода ASCII, а атрибут задает, каким образом данный символ будет выведен на экран (его цвет, яркость, и т.д.). Borland C++ предоставляет полный набор подпрограмм для манипулирования текстовым экраном, для вывода текста

непосредственно на экран и управления атрибутами ячеек.

- В графическом режиме экран компьютера делится на элементы изображения (пикселы); каждый элемент изображения представляет собой отображение на экране одной точки. Число элементов изображения на экране (т.е. его разрешающая способность) зависит от типа подключенного к вашей системе видеоадаптера и режима, в который установлен этот адаптер. Для получения на экране графических изображений Borland C++ предоставляет библиотеку графических функций: вы можете создавать на экране линии и формы, заполненные шаблонами замкнутые области, а также управлять цветом каждого элемента изображения.

В текстовом режиме позиция верхнего левого угла экрана определяется координатами (1,1), где **x**-координата растет слева-направо, а **y**-координата увеличивается сверху-вниз. В графическом режиме позиция верхнего левого угла определяется координатами (0,0), с теми же направления возрастания координат.

Текстовые и графические окна

Borland C++ обеспечивает функции для создания окон и управления ими в текстовом режиме (и графических окон в графическом режиме).

Окно представляет собой прямоугольную область, определенную на видеоэкране вашего компьютера PC, когда он находится в текстовом режиме. Когда ваша программа выполняет вывод на экран, то область вывода будет в таком случае ограничена активным окном. Остальная часть экрана (вне окна) остается без изменений.

По умолчанию размер окна равен всему экрану. При помощи функции **window** ваша программа может изменить данное использование по умолчанию полноэкранного текстового окна на текстовое окно, меньшее, чем полный экран. Эта функция задает позицию окна в экранных координатах.

В графическом режиме вы также можете определить некоторую прямоугольную область экрана PC. Эта область называется графическим окном или областью просмотра

(**viewport**). Когда ваша графическая программа выполняет вывод рисунков и т.д., графическое окно действует как виртуальный экран. Остальная часть экрана (вне графического окна) остается без изменений. Определить графическое окно можно через экранные координаты, вызвав функцию **setviewport**.

За исключением функций определения текстовых и графических окон, все остальные функции, как текстового, так и графического режимов, даются в локальных координатах активного текстового или графического окна, а не в абсолютных экранных координатах. При этом верхний левый угол текстового окна будет представлять собой начало координат (1,1). В графическом режиме начало координат графического окна будет равно (0,0).

Программирование в графическом режиме

Borland C++ имеет отдельную библиотеку с более чем 70 графическими функциями, начиная от функций высокого уровня (таких как **setviewport**, **bar3d** и **drawpoly**) и кончая бит-ориентированными функциями (типа **getimage** и **putimage**). Графическая библиотека поддерживает многочисленные типы линий и заполнителей, а также предоставляют вам различные текстовые шрифты, которые вы можете изменять по размерам, способу выравнивания, а также ориентировать их либо по горизонтали, либо по вертикали.

Эти функции находятся в библиотечном файле **GRAPHICS.LIB**, а их прототипы — в файле заголовка **graphics.h**. Кроме этих двух файлов, в состав графического пакета входят драйверы графических устройств (файлы ***.BGI**) и символьные шрифты (файлы ***.CHR**).

Если вы используете компилятор **BCC.EXE**, нужно в командной строке указать библиотеку **GRAPHICS.LIB**. Например, если ваша программа, **MYPROG.C**, использует графику, то командная строка компилятора **BCC** должна иметь вид:

```
BCC MYPROG GRAPHICS.LIB
```

При построении программы компоновщик автоматически компонует графическую библиотеку C++.

Поскольку графические функции используют указатели **far**, графика в случае модели памяти **tiny** не поддерживается.

Графическая библиотека только одна и не имеет версий по моделям памяти (по сравнению со стандартными библиотеками **CS.LIB**, **CC.LIB**, **CM.LIB** и т.д., которые зависят от используемой модели памяти). Каждая функция в файле **GRAPHICS.LIB** является **far** (дальней) функцией, а графические функции, использующие указатели работают с дальними указателями. Для правильной работы графических функций в каждом использующем графические функции модуле требуется директива **#include graphics.h**.

Функции библиотеки graphics

Графические функции Borland C++ делятся на несколько категорий:

- функции управления графической системой;
- функции черчения и заполнения;
- функции манипулирования экранами и графическими окнами;
- функции вывода текстов;
- функции управления цветами;
- функции обработки ошибок;
- функции запроса состояния.

Управление графической системой

Ниже приводится краткое перечисление всех функций управления графической системой:

closegraph

Закрывает графическую систему.

detectgraph

Проверяет аппаратное обеспечение и определяет, какие графические драйверы использовать; рекомендует предпочтительный режим.

graphdefaults

Сбрасывает все переменные графической системы в значения по умолчанию.

_graphfreemem

Отменяет выделенную графике память. Используется для определения собственной подпрограммы.

_graphgetmem

Распределяет память графике; используется для определения собственной подпрограммы.

getgraphmode

Возвращает текущий графический режим.

getmoderange

Возвращает минимальный и максимальный допустимые режимы для заданного драйвера.

initgraph

Инициализирует графическую систему и переводит аппаратное обеспечение в графический режим.

installuserdriver

Устанавливает дополнительный драйвер устройства в таблице драйверов устройства BGI.

installuserfont

Загружает поставляемый файл векторного (штрихового) шрифта в таблицу символьных файлов BGI.

registerbgldriver

Регистрирует внешний или загруженный пользователем файл драйвера для включения во время компоновки.

restorecrtmode

Восстанавливает первоначальный (существовавший до Initgraph) режим экрана.

setgraphbufsize

Задает размер внутреннего графического буфера.

setgraphmode

Выбирает заданный графический режим, очищает экран и восстанавливает все умолчания.

Графический пакет компилятора Borland C++ обеспечивает графические драйверы для следующих графических адаптеров (и полностью совместимых с ними):

- Цветной/графический адаптер (CGA);
- Многоцветная графическая матрица (MCGA);
- Улучшенный графический адаптер (EGA);
- Видеографическая матрица (VGA);
- Графический адаптер Hercules;
- Графический адаптер серии AT&T 400;
- Графический адаптер 3270 PC;
- Графический адаптер IBM 8514.

Для запуска графической системы вы должны прежде всего вызвать функцию **initgraph**. Функция **initgraph** загружает графический драйвер и переводит систему в графический режим.

Вы можете указать для функции **initgraph** использование конкретного графического драйвера и конкретный режим, либо задать автообнаружение установленного видеоадаптера и выбор соответственного драйвера уже во время выполнения. Если вы задали в функции **initgraph** автообнаружение, то она сама вызовет функцию **detectgraph** для выбора графического драйвера и режима. Если вы задали в **initgraph** использование конкретного графического драйвера и режима, то вы сами отвечаете за физическое присутствие соответствующего аппаратного обеспечения. Если заставить **initgraph** пытаться использовать отсутствующее аппаратное обеспечение, то результат в таком случае непредсказуем.

После того, как графический драйвер загружен, вы можете определить его имя при помощи функции **getdrivename**, а число поддерживаемых драйвером режимов — при помощи функции **getmaxmode**. Функция **getgraphmode** сообщит вам, в каком графическом режиме вы находитесь в текущий момент. Имея номер режима, вы можете определить его имя при помощи функции **getmodename**. Вы также имеете возможность изменить графический режим при помощи функции **setgraphmode** и вернуть исходный видеорежим (тот, который был установлен до инициализации графики) с помощью **restorecrtmode**. Функция **restorecrtmode** вернет экран в текстовый режим, но не закроет при этом графическую систему (загруженные шрифты и драйверы останутся в памяти).

Функция **graphdefaults** сбрасывает установки состояния графической системы (размеры графического окна, цвет линий, цвет и шаблон заполнителя и т.д.) в исходное состояние. Функции **installuserdriver** и **installuserfont** позволяют установить в графической системе новые драйверы устройства и шрифты.

И наконец, закончив работу в графике, вы должны вызвать функцию **closegraph** для того, чтобы закрыть графическую систему. Функция **closegraph** выгружает драйвер из памяти и восстанавливает первоначальный видеорежим (через обращение к **restorecrtmode**).

Обычно подпрограмма **initgraph** загружает графический драйвер, распределяя для этого драйвера память и затем загружая туда с диска соответствующий файл **.BGI**. В качестве альтернативы данной схеме динамической загрузки вы можете скомпоновать нужный файл графического драйвера (или несколько таких файлов) непосредственно с файлом выполняемой программы. Для этого файл **.BGI** сначала преобразуется в файл **.OBJ** (при помощи утилиты **BGIOBJ**), после чего в исходный код помещается вызов функции **registerbgidriver** (до вызова **initgraph**), чтобы зарегистрировать графический драйвер(ы) в системе. При построении программы вы должны выполнить компоновку файлов **.OBJ** всех зарегистрированных драйверов.

После определения того, какой графический драйвер должен использоваться (посредством **detectgraph**) функция **initgraph** проверяет, был ли желаемый драйвер зарегистрирован. Если был, то **initgraph** обращается к зарегистрированному драйверу непосредственно в памяти. В противном случае функция **initgraph** распределяет память для драйвера и загружает нужный файл **.BGI** с диска.

Использование функции **registerbgidriver** относится к более сложным методам программирования, не рекомендуемым для начинающих программистов.

Во время выполнения графической системе может понадобиться распределить память для драйверов, шрифтов и внутренних буферов. При необходимости она вызывает функцию **_graphgetmem** для распределения памяти и функцию **_graphfreemem** для ее освобождения. По умолчанию данные

подпрограммы просто вызывают функции **malloc** и **free**, соответственно.

Действие этих функций по умолчанию можно переопределить, определив собственные функции **_graphgetmem** и **_graphfreemem**. Благодаря этому вы можете сами управлять распределением памяти для графики. Однако, ваши варианты функций управления распределением памяти должны иметь те же имена: они заменят собой используемые по умолчанию функции с теми же именами из стандартных библиотек языка Си.

Определив собственные функции **_graphgetmem** и **_graphfreemem**, вы можете получить предупреждение «**duplicate symbols**» («повторение символических имен»). Это предупреждение можно игнорировать.

Черчение и заполнение

Ниже приводится краткий обзор функций черчения и закраски:

Функции черчения

arc

Чертит дугу окружности.

circle

Чертит окружность.

drawpoly

Чертит контур многоугольника.

ellipse

Чертит эллиптическую дугу.

getarccoords

Возвращает координаты последнего вызова **arc** или **ellipse**.

getaspectratio

Возвращает коэффициент сжатия для текущего графического режима.

getlinesettings

Возвращает текущий тип линии, шаблон линии и толщину линии.

line

Чертит линию из точки (x0,y0) в (x1,y1).

linerel

Чертит линию в точку, задаваемую относительным расстоянием от текущей позиции (CP).

lineto

Чертит линию из текущей позиции (CP) в (x,y).

moveto

Перемещает текущую позицию (CP) в (x,y).

moverel

Перемещает текущую позицию (CP) на относительное расстояние.

rectangle

Рисует прямоугольник.

setaspectratio

Изменяет коэффициент сжатия по умолчанию.

setlinestyle

Устанавливает толщину и тип текущей линии.

Функции закраски

bar

Чертит и закрашивает столбец.

bar3d

Чертит и закрашивает трехмерный столбец.

fillellipse

Чертит и закрашивает эллипс.

fillpoly

Чертит и закрашивает многоугольник.

getfillpattern

Возвращает определяемый пользователем шаблон закраски.

getfillsettings

Возвращает информацию о текущем шаблоне и цвете закраски.

pieslice

Чертит и закрашивает сектор окружности.

sector

Чертит и закрашивает эллиптический сектор.

setfillpattern

Выбирает шаблон закраски, определяемый пользователем.

setfillstyle

Устанавливает шаблон и цвет закраски.

При помощи функций черчения и раскрашивания Borland C++ вы можете вычерчивать цветные линии, дуги, окружности, эллипсы, прямоугольники, секторы, дву- и трехмерные столбики, многоугольники, а также различные правильные или неправильные формы, являющиеся комбинациями перечисленных графических примитивов. Ограниченную форму изнутри или снаружи можно заполнить одним из 11 предопределенных шаблонов (образцов заполнителей), либо шаблоном, определенным пользователем. Можно также управлять толщиной и стилем линии вычерчивания, а также местоположением текущей позиции (**CP**).

Линии и незакрашенные формы вычерчиваются при помощи функций **arc**, **circle**, **drawpoly**, **ellipse**, **line**, **linerel**, **lineto** и **rectangle**. Затем можно закрасить эти формы с помощью **floodfil**, либо можно объединить вычерчивание/закраску в одном шаге при помощи функций **bar**, **bar3d**, **fillellipse**, **fillpoly**, **pieslice** и **sector**. Функция **setlinestyle** позволяет задать тип линий (и граничных линий форм): толстая или тонкая, сплошная, пунктир и т.д., либо для вычерчивания линии можно задать ваш собственный шаблон. При помощи функции **setfillstyle** можно выбрать предопределенный шаблон заполнения, либо определить собственный шаблон заполнения в **setfillpattern**. Функция **moveto** позволяет переместить **CP** в желаемую позицию, а функция **moverel** позволяет сдвинуть ее на желаемую величину смещения.

Выяснить текущий тип и толщину линии позволяет функция **getlinesettings**. Информацию о текущем шаблоне заполнения и цвете заполнителя можно получить через функцию **getfillsettings**. Определяемый пользователем шаблон заполнения можно получить при помощи функции **getfillpattern**.

Получить сведения о коэффициенте относительного удлинения (коэффициенте масштабирования, применяемом графической системой для того, чтобы окружности выглядели

круглыми) позволяет функция **getaspectratio**, а получить координаты последней нарисованной дуги или эллипса — функция **getarccoords**. Если окружности не получаются идеально круглыми, можно исправить дело при помощи функции **setaspectratio**.

Манипулирование экраном и графическими окнами

Ниже приводится краткий обзор функций манипулирования с экраном, графическими окнами, битовыми образами и элементами изображения.

Функции работы с экраном

cleardevice

Очищает экран (активную страницу).

setactivepage

Устанавливает активную страницу для графического вывода.

setvisualpage

Устанавливает номер видимой графической страницы.

Функции работы с графическими окнами

clearviewport

Очищает текущее графическое окно.

getviewsettings

Возвращает информацию о текущем графическом окне.

setviewport

Устанавливает текущее графическое окно для направления на него графического вывода.

Функции работы с битовыми образами

getimage

Записывает битовый образ в заданный участок памяти.

imagesize

Возвращает число байт, требуемых для хранения некоторой прямоугольной области экрана.

putimage

Помещает на экран ранее записанный в память битовый образ.

Функции работы с элементами изображения

getpixel

Получает цвет элемента изображения в (x,y).

putpixel

Помещает элемент изображения на экран в точку (x,y).

Помимо черчения и закрашивания, графическая библиотека предлагает несколько функций для манипулирования экраном, графическими окнами, образами и указателями. Вызвав функцию **cleardevice**, можно сразу очистить весь экран. Данная подпрограмма стирает экран и помещает текущую позицию в графическое окно, но при этом оставляет действующими все прочие установки графической системы (типы линии, заполнения и текста; раскраска, установки графического окна и т.д.).

В зависимости от имеющегося у вас графического адаптера, ваша система может иметь от одного до четырех буферов экранных страниц, представляющих собой области памяти, где хранится информация по точкам о конкретных полноэкранных образах. При помощи функций **setactivepage** и **setvisualpage**, соответственно, вы можете указать активную страницу экрана (т.е. куда будет направлен вывод графических функций), и визуальную (отображаемую) страницу экрана (т.е. страницу, находящуюся в текущий момент на дисплее).

Когда ваш экран находится в графическом режиме, с помощью функции **setviewport** вы можете определить графическое окно (или прямоугольное «виртуальное окно») на экране. Позиция графического окна задается в абсолютных экранных координатах. Кроме того, задается активное или неактивное состояние функции «отсечения». Очистка графического окна выполняется при помощи функции **clearviewport**. Для того, чтобы получить абсолютные экранные координаты и состояние «отсечения», следует воспользоваться функцией **getviewsettings**.

Можно взять часть экранного образа при помощи функции **getimage**, вызвать **imagesize** для вычисления числа байт для хранения этого образа в памяти, а затем вернуть образ на экран (в любую желаемую позицию) с помощью функции **putimage**. Координаты всех функций вывода (черчения, заполнения, тексты и т.д.) зависят от выбранного графического окна.

Благодаря функциям **getpixel** (возвращающей цвет данного элемента изображения) и **putpixel** (которая отображает данный элемент изображения на экране заданным цветом) можно также манипулировать цветом отдельных элементов изображения.

Текстовый вывод в графическом режиме

Ниже приводится краткое описание функций текстового вывода в графическом режиме:

gettextsettings

Возвращает текущий текстовый шрифт, направление, размер и выравнивание.

outtext

Посылает строку на экран в текущую позицию (**CP**).

outtextxy

Посылает текст на экран в заданную позицию.

registerbgifont

Регистрирует компонуемый или определяемый пользователем шрифт.

settextjustify

Устанавливает значения выравнивания текста, используемые функциями **outtext** и **outtextxy**.

settextstyle

Устанавливает шрифт, тип и коэффициент увеличения текущего текста.

setusercharsize

Устанавливает соотношение между высотой и шириной штриховых шрифтов.

textheight

Возвращает высоту строки в элементах изображения.

textwidth

Возвращает ширину строки в элементах изображения.

Графическая библиотека включает в себя матричный шрифт 8x8 и несколько векторных шрифтов для вывода текста в графическом режиме.

В матричном битовом шрифте каждый символ определяется как матрица элементов изображения.

В векторном шрифте каждый символ определяется как последовательность векторов, сообщающих графической системе, как создается данный символ.

Преимущество использования векторных шрифтов становится очевидным, когда вы начинаете рисовать большие по размеру символы. Поскольку векторный шрифт определяется как последовательность векторов, при увеличении размера он сохранит хорошее разрешение и качество изображения. И напротив, когда вы увеличиваете битовый матричный шрифт, матрица умножается на соответствующий коэффициент масштабирования. Чем больше этот коэффициент, тем хуже становится разрешение символов. Для малых размеров такой вид шрифта вполне удовлетворителен, однако для больших размеров вам лучше выбрать векторный шрифт.

В графике текст выводится функциями outtext или outtextxy, а управление его выравниванием (относительно текущей позиции) выполняет функция **settextjustify**. При помощи функции **settextstyle** вы должны выбрать символьный шрифт, направление его размещения (горизонтальное или вертикальное) и размер (масштаб). Узнать текущие установки вывода текстов можно при помощи функции **gettextsettings**, которая возвращает текущий текстовый шрифт, выравнивание, увеличение и направление в структуре **textsettings**. Функция **setusercharsize** позволяет модифицировать ширину и высоту векторных шрифтов.

Если средство отсечения изображения включено, то выводимые функциями **outtext** и **outtextxy** текстовые строки будут отсекаться по границам графического окна. Если отсечение отключено, то тексты с матричным шрифтом, символы которых не помещаются целиком в окне, отбрасываются полностью. В случае же векторных шрифтов не поместившиеся тексты просто отсекаются по границе окна.

Для того, чтобы определить экранный размер данной текстовой строки, вызовите функцию **textheight** (которая измеряет высоту текста в элементах изображения) и **textwidth** (измеряющую его ширину в элементах изображения).

По умолчанию битовый матричный шрифт 8x8 встроен в графический пакет и поэтому всегда доступен во время выполнения. Векторные шрифты все хранятся в отдельных файлах **.CHR**. Они могут загружаться во время выполнения или преобразовываться в файлы **.OBJ** (при помощи утилиты **BGIOBJ**) и затем компоноваться с вашим файлом **.EXE**.

Обычно подпрограмма **settextstyle** загружает файл шрифта, распределяя память для него и затем загружая с диска соответствующий **.CHR-файл**. В качестве альтернативы данной схеме динамической загрузки вы можете скомпоновать файл шрифта (или несколько таких файлов) непосредственно с выполняемым файлом программы. Для этого сначала требуется преобразовать файл **.CHR** в файл **.OBJ** (с помощью утилиты **BGIOBJ**, а затем поместить в исходную программу вызовы **registerbgifont** (перед вызовом функции **settextstyle)** для того, чтобы зарегистрировать данный символьный шрифт(ы). При построении программы для всех зарегистрированных вами векторных шрифтов необходимо скомпоновать полученные файлы **.OBJ**.

Использование функции **registerbgifont** относится к сложным методам программирования и не рекомендуется начинающим программистам.

Управление цветом

Ниже приводится краткое описание функций для управления цветом изображений:

Функции получения информации о цвете

getbcolor

Возвращает текущий цвет фона.

getcolor

Возвращает текущий цвет вычерчивания.

getdefaultpalette

Возвращает структуру определения палитры.

getmaxcolor

Возвращает максимальное значение цвета доступное в текущем графическом режиме.

getpalette

Возвращает текущую палитру и ее размер.

getpalettesize

Возвращает размер просмотровой таблицы палитры.

Функции установки одного или более цветов

setallpalette

Изменяет все цвета палитры, как задано.

setbkcolor

Устанавливает текущий цвет фона

setcolor

Устанавливает текущий цвет вычерчивания.

setpalette

Изменяет один из цветов палитры, как указано ее аргументами.

Прежде чем перейти к рассмотрению работы функций управления цветом изображения, дадим базовое описание того, как эти цвета фактически получаются на вашем графическом экране.

Элементы изображения и палитры

Графический экран представляет собой массив элементов изображения. Каждый элемент изображения соответствует одной (цветной) точке на экране. Значение элемента изображения не задает точный цвет этой точки напрямую; на самом деле это некоторый индекс таблицы цветов, называемой палитрой. Каждый элемент палитры, соответствующий данному значению элемента изображения, содержит точную информацию о цвете, которым будет отображен этот элемент изображения.

Такая схема косвенных обращений имеет множество следствий. Хотя аппаратное обеспечение может позволять отображение множества цветов, одновременно на экране может находиться только некоторое их подмножество. Количество одновременно находящихся на экране цветов равно числу элементов палитры (размеру палитры). Например, EGA позволяет наличие 64 цветов, но лишь 16 из них может находиться на экране сразу; таким образом, размер палитры EGA равен 16.

Размер палитры определяет диапазон значений, которые может принимать элемент изображения, от 0 до (размер-1). Функция **getmaxcolor** возвращает максимальное допустимое значение элемента изображения (размер-1) для текущего графического драйвера и режима.

При обсуждении графических функций Borland C++ мы часто используем термин «цвет», например текущий цвет вычерчивания, цвет заполнения и цвет элемента изображения. Фактически цветом мы здесь называем значение элемента изображения: это некоторый индекс в палитре. Только палитра реально определяет фактический цвет на экране. Манипулируя палитрой, вы можете изменять фактические цвета, выводимые на дисплей, даже хотя значения элементов изображения (цвета вычерчивания, заполнения и т.д.) могут не изменяться.

Цвет фона и вычерчивания

Цвет фона всегда соответствует значению элемента изображения 0. Когда выполняется очистка области экрана в цвет фона, это означает просто установку всех элементов изображения этой области в значение 0.

Цветом вычерчивания (цветом переднего плана) называется значение, в которое устанавливаются элементы изображения при вычерчивании линий. Цвет вычерчивания устанавливается функцией setcolor(n), где n есть допустимое для текущей палитры значение элемента изображения

Управление цветом на CGA

Из-за различий в графическом аппаратном обеспечении фактическое управление цветами различно для CGA и EGA, что заставляет нас рассмотреть их по отдельности. Управление цветом для драйвера AT&T, а также режимы низкой разрешающей способности драйвера MCGA аналогичны управлению цветом CGA.

В случае адаптера CGA вы можете выбрать либо режим низкой разрешающей способности (320x200), который допускает использование четырех цветов, либо режим высокой разрешающей способностей (640x200), где допускается использование двух цветов.

CGA в режиме низкой разрешающей способности

В режиме низкой разрешающей способности вы имеете возможность выбрать одну из четырех четырехцветных палитр. В каждой из этих четырех палитр вы можете сами установить только первый (цвет 0) элемент; цвета 1, 2 и 3 являются фиксированными. Первый элемент палитры (цвет 0) — это цвет фона. Этот цвет может являться одним из 16 имеющихся цветов (см. таблицу цветов фона, приводимую ниже).

Вы выбираете желаемую палитру, выбирая соответствующий режим (CGAC0, CGAC1, CGAC2, CGAC3); эти режимы используют палитры цветов от 0 до 3, соответственно, как показано в следующей таблице. Цвета вычерчивания в CGA и эквивалентные им константы определяются в **graphics.h**.

Константа, присвоенная номеру цвета (значению эл. изображения)

Номер палитры	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

Для того, чтобы назначить один из этих цветов цветом вычерчивания **CGA**, нужно вызвать функцию **setcolor**, задав в ней в качестве аргумента либо номер цвета, либо имя соответствующей константы; например, если вы используете палитру 3 и желаете назначить цветом вычерчивания **cyan**, то можно записать:

```
setcolor(1);
```

или

```
setcolor(CGA_CYAN);
```

В следующей таблице перечислены назначаемые для **CGA** цвета фона:

Числовое значение	Символическое имя
0	BLACK
8	DARKGRAY
1	BLUE
9	LIGHTBLUE
2	GREEN

10	**LIGHTGREEN**
3	**CYAN**
11	**LIGTHCYAN**
4	**RED**
12	**LIGHTRED**
5	**MAGENTA**
13	**LIGHTMAGENTA**
6	**BROWN**
14	**YELLOW**
7	**LIGHTGRAY**
15	**WHITE**

Цвета CGA для переднего плана те же, что находятся в данной таблице. Для назначения одного из этих цветов в качестве фонового цвета служит функция **setbkcolor(цвет)**, где **цвет** — это один из элементов приведенной выше таблицы. Отметим, что для CGA цвет не является значением элемента изображения (индексом в палитре). Он прямо задает фактический цвет, помещаемый в первый элемент палитры.

CGA в режиме высокой разрешающей способности

В режиме высокой разрешающей способности (640x200) CGA работает с двумя цветами — черным цветом фона и цветным передним планом. Элементы изображения могут принимать при этом значения только 0 или 1. В связи с особенностями CGA цветом переднего плана фактически является тот цвет, который аппаратное обеспечение считает цветом фона. Таким образом, цвет переднего плана устанавливается подпрограммой **setbkcolor**.

Цвет для переднего плана может быть выбран из предыдущей таблицы. CGA далее будет использовать этот цвет для отображения им всех элементов изображения, имеющих значение 1.

Режимы **CGAHI**, **MCGAMED**, **MCGAHI**, **ATT400MED** и **ATT400HI** работают аналогичным образом.

Подпрограммы управления палитрой в случае CGA

Поскольку палитра CGA является предопределенной, подпрограмму **setallpalette** использовать в данном случае нельзя. Также не следует использовать **setpalette(индекс, фактический_цвет)**, за исключением **индекс=0**. (Это альтернативный способ установки фонового цвета CGA равным фактическому цвету)

Управление цветом для EGA и VGA

В случае EGA палитра содержит 16 элементов из общего количества 64 возможных цветов, причем каждый из элементов палитры может быть задан пользователем. Доступ к текущей палитре выполняется через функцию **getpalette**, которая заполняет структуру, включающую в себя размер палитры (16) и массив фактических элементов палитры («аппаратные номера цветов», хранимые в палитре). Элементы палитры можно изменять как по отдельности при помощи **setpalette**, либо все сразу через функцию **setallpalette**.

Палитра EGA по умолчанию соответствует 16 цветам CGA, которые были даны в предыдущей таблице цветов: черный равен элементу 0, голубой равен элементу 1, ... , белый равен элементу 15. В **graphics.h** определены константы, которые содержат соответствующие цветам аппаратные значения: это **EGA_BLACK**, **EGA_WHITE** и т.д. Эти значения могут быть также получены через функцию **getpalette**.

Подпрограмма **setbkcolor(цвет)** на EGA работает несколько иначе, чем на CGA. На EGA **setbkcolor** копирует фактическое значение цвета, хранящееся в элементе **#цвет**, в элемент **#0**.

Что касается цветов, то драйвер VGA работает фактически так же, как и драйвер EGA; он просто имеет более высокое разрешение (и меньшие по размеру элементы изображения)

Обработка ошибок в графическом режиме

Ниже приведены функции обработки ошибок в графическом режиме:

grapherrormsg

Возвращает строку с сообщением об ошибке для заданного кода ошибки.

graphresult

Возвращает код ошибки для последней графической операции, в которой встретилась ошибка.

Если ошибка произошла при вызове графической библиотечной функции (например, не найден шрифт, запрошенный функцией **settextstyle**), устанавливается внутренний код ошибки. Доступ к коду ошибки для последней графической операции, сообщившей об ошибке, выполняется при помощи функции **graphresult**. Вызов **grapherrormsg(graphresult())** возвращает строку сообщения об ошибке из приведенных выше.

Код возврата ошибки накапливается, изменяясь только когда графическая функция сообщает об ошибке. Код возврата ошибки сбрасывается в 0 только при успешном выполнении **initgraph**, либо при вызове функции **graphresult**. Таким образом, если вы хотите знать, какая графическая функция возвратила ошибку, нужно хранить значение **graphresult** во временной переменной и затем проверять ее.

Код ошибки, константа графической ошибки и соответствующая строка с сообщением об ошибке приведены ниже:

0
grOk

No error (нет ошибки)

-1
grNoInitGraph

(BGI) graphics not installed (use initgraph) (графика не инсталлирована используйте функцию initgraph)

-2
grNotDetected

Graphics hardware not detecte (графическое аппаратное обеспечение не обнаружено)

-3
grFileNotFound

Device driver file not found (не найден файл драйвера устройства)

-4
grInvalidDriver

Invalid device driver file (неверный файл драйвера устройства)

-5
grNoLoadMem

Not enough memory to load driver (не хватает памяти для загрузки драйвера)

-6
grNoScanMem

Out of memory in scan fill (кончилась память при сканирующем заполнении)

-7
grNofloodMem

Out of memory in flood fill (кончилась память при лавинном заполнении)

-8
grFontNotFound

Font file not found (файл шрифта не найден)

-9
grNoFontMem

Not enough memory to load font (не хватает памяти для загрузки шрифта)

-10
grInvalidMode

Invalid graphics mode for selected driver (недопустимый графический режим для выбранного драйвера)

-11
grError

Graphics error (графическая ошибка)

-12
grIOerror

Graphics I/O error (графическая ошибка ввода-вывода)

-13
grInvalidFont

Invalid font file (неверный файл шрифта)

-14
grInvalidFontNum

Invalid font number (неверный номер шрифта)

-15
grInvalidDeviceNum

Invalid device number (неверный номер устройства)

-18
grInvalidVersion
Invalid version of file (неправильная версия файла)

Функции запроса состояния

Ниже приводится краткое изложение функций запроса состояния графического режима:

Функции запроса состояния графического режима

getarccoords
Возвращает информацию о координатах, заданных в последнем вызове **arc** или **ellipse**.

getaspectratio
Возвращает коэффициент сжатия для графического экрана.

getbkcolor
Возвращает текущий цвет фона.

getcolor
Возвращает текущий цвет вычерчивания.

getdrivername
Возвращает имя текущего графического драйвера.

getfillpattern
Возвращает шаблон заполнения, определяемый пользователем.

getfillsettings
Возвращает информацию о текущем шаблоне и цвете заполнения.

getgraphmode
Возвращает текущий графический режим.

getlinesettings
Возвращает текущие стиль, шаблон и толщину линии.

getmaxcolor
Возвращает максимально допустимое на текущий момент значение элемента изображения.

getmaxmode
Возвращает максимально допустимый номер режима для текущего драйвера.

getmaxx

Возвращает текущее разрешение по оси **x**.

getmaxy

Возвращает текущее разрешение по оси **y**.

getmodename

Возвращает имя данного режима драйвера.

getmoderange

Возвращает диапазон режимов для данного драйвера.

getpalette

Возвращает текущую палитру и ее размер.

getpixel

Возвращает цвет элемента изображения в (**x,y**).

gettextsettings

Возвращает текущий шрифт, направление, размер и способ выравнивания текста.

getviewsettings

Возвращает информацию о текущем графическом окне.

getx

Возвращает координату x текущей позиции (**CP**).

gety

Возвращает координату y текущей позиции (**CP**).

В каждой из категорий графических функций Borland C++ имеется хотя бы одна функция запроса состояния. Каждая из графических функций запроса состояния Borland C++ имеет имя вида «**get *что-то***» (за исключением категории функций обработки ошибок). Некоторые из них не воспринимают никаких аргументов и возвращают единственное значение, представляющее собой искомую информацию. Прочие считывают указатель структуры, определяемой в файле **graphics.h**, заполняют эту структуру соответствующей информацией и не возвращают никаких значений.

Функциями запроса состояния категории управления графической системы являются **getgraphmode**, **getmaxmode** и **getmoderange**. Первая из них возвращает целое число, определяющее текущий графический драйвер и режим, вторая

возвращает максимальный номер режима для этого драйвера, а третья возвращает диапазон режимов, поддерживаемых данным графическим драйвером. **getmaxx** и **getmaxy** возвращают соответственно максимальные экранные координаты **x** и **y** для текущего графического режима.

Функциями запроса состояния категории вычерчивания и заполнения являются функции **getarccoords**, **getaspectratio**, **getfillpattern** и **getlinesettings**. Функция **getarccoords** заполняет структуру, содержащую координаты, которые использовались при последнем вызове функций **arc** или **ellipse**. Функция **getaspectratio** сообщает текущий коэффициент сжатия, используемый графической системой для того, чтобы окружности выглядели круглыми. Функция **getfillpattern** возвращает текущий определяемый пользователем шаблон заполнения. Функция **getfillsettings** заполняет некоторую структуру текущим шаблоном и цветом заполнения. Функция **getlinesettings** заполняет структуру текущим стилем линии (сплошная, пунктир и т.д.), толщиной (обычная или увеличенная), а также шаблоном линии.

Функциями запроса состояния категории манипулирования графическим окном являются **getviewsettings**, **getx**, **gety** и **getpixel**. После того, как графическое окно определено, вы можете найти его абсолютные экранные координаты и выяснить состояние режима отсечения, вызвав **getwiewsettings**, которая заполняет соответствующей информацией некоторую структуру. Функции **getx** и **gety** возвращают (относительно графического окна) **x**- и **y**-координаты текущей позиции (**CP**). Функция **getpixel** возвращает цвет указанного элемента изображения.

Функция запроса состояния категории вывода текста в графическом режиме имеется только одна, и притом всеобъемлющая, — **gettextsettings**. Эта функция заполняет структуру информацией о текущем символьном шрифте, направлении вывода текста (по горизонтали или по вертикали), коэффициенте увеличения символов, а также виде выравнивания (как для горизонтально, так и для вертикально-ориентированных текстов).

Функциями запроса состоянии категории управления цветом Borland C++ являются функция **getbkcolor**, возвращающая текущий цвет фона, функция **getcolor**, возвращающая текущий цвет вычерчивания и функция **getpalette**, заполняющая структуру,

которая включает в себя размер текущей палитры и ее содержимое. Функция **getmaxcolor** возвращает максимально допустимое значение элемента изображения для текущего графического драйвера и режима (размер палитры -1).

И наконец, функции **getmodename** и **getdrivername** возвращают имя заданного режима драйвера и имя текущего графического драйвера, соответственно.

Библиотеки DOS

Ниже представлен краткий обзор библиотечных программ Borland C++, доступных только для 16-разрядных приложений DOS. Библиотечные подпрограммы состоят из функций и макрокоманд, которые можно вызывать в программах Си и C++ для выполнения различных задач, включая ввод-вывод различного уровня, работу со строками и файлами, распределение памяти, управление процессом, преобразование данных, математические вычисления и др.

Библиотеки исполняющей системы

В приложениях DOS используются статические библиотеки исполняющей системы (**OBJ** и **LIB**).

Существует несколько версий библиотеки исполняющей системы. Это версии для конкретных моделей памяти и диагностические библиотеки. Имеются также дополнительные библиотеки, обеспечивающие контейнеры, графику и математические операции. При выборе используемых библиотек исполняющей системы следует иметь в виду что перечисленные ниже библиотеки используются только в 16-разрядных приложениях DOS.

Библиотеки поддержки DOS

Статические (**OBJ** и **LIB**) 16-разрядные библиотеки исполняющей системы Borland C++ после установки записываются в подкаталог **LIB**. Для каждого из имен этих библиотек символ «**?**» представляет одну и 6 поддерживаемых Borland моделей памяти. Каждая модель имеет собственный библиотечный файл и файл поддержки математических операций с версиями подпрограмм, написанных для конкретной модели.

В следующей таблице перечислены имена библиотек Borland C++, которые доступны только для 16-разрядных приложений DOS.

BIDSH.LIB

Библиотеки классов Borland модели памяти huge.

BIDSDBH.LIB

Диагностическая версия той же библиотеки.

C?.LIB

Библиотеки DOS.

C0F.OBJ

MS-совместимые библиотеки запуска.

C0?.OBJ

Библиотеки запуска BC.

EMU.LIB

Эмуляция операций с плавающей точкой.

FP87.LIB

Для программ, работающих на машинах с сопроцессором 80x87.

GRAPHICS.LIB

Графический интерфейс Borland.

MATH?.LIB

Математические подпрограммы.

OVERLAY.LIB

Разработка оверлеев.

Графические подпрограммы

Следующие подпрограммы позволяют создавать экранные графические представления с текстовой частью.

- arc (graphics.h)
- bar (graphics.h)
- bar3d (graphics.h)
- circle (graphics.h)
- cleardevice (graphics.h)
- clearviewport (graphics.h)

- closgraph (graphics.h)
- detectgraph (graphics.h)
- drawpoly (graphics.h)
- ellipse (graphics.h)
- fillellipse (graphics.h)
- fillpoly (graphics.h)
- floofill (graphics.h)
- getfillsettings (graphics.h)
- getgraphmode (graphics.h)
- getimage (graphics.h)
- getfinesettings (graphics.h)
- getmaxcolor (graphics.h)
- getmaxmode (graphics.h)
- getmaxx (graphics.h)
- getmaxy (graphics.h)
- getmodename (graphics.h)
- getmoderange (graphics.h)
- getpalette (graphics.h)
- getpixel (graphics.h)
- gettextsettings (graphics.h)
- getviewsettings (graphics.h)
- getx (graphics.h)
- gety (graphics.h)
- graphdefaults (graphics.h)
- grapherrormsg (graphics.h)
- _graphfreemem (graphics.h)
- _graphgetmem (graphics.h)
- graphresult (graphics.h)
- getarccoords (graphics.h)
- getaspectratio (graphics.h)

- getbkcolor (graphics.h)
- getcolor (graphics.h)
- getdefaultpallette (graphics.h)
- getdrivername (graphics.h)
- getfillpattern (graphics.h)
- imagesize (graphics.h)
- initgraph (graphics.h)
- installuserdriver (graphics.h)
- installuserfont (graphics.h)
- line (graphics.h)
- linerel (graphics.h)
- lineto (graphics.h)
- moverei (graphics.h)
- moveto (graphics.h)
- outtext (graphics.h)
- outtextxy (graphics.h)
- pieslice (graphics.h)
- pufimage (graphics.h)
- pulpixel (graphics.h)
- rectangle (graphics.h)
- registerbgidriver (graphics.h)
- registerbgifont (graphics.h)
- restorecrtmode (graphics.h)
- sector (graphics.h)
- settaffpalette (graphics.h)
- setaspectratio (graphics.h)
- setbkcolor (graphics.h)
- setcolor (graphics.h)
- setcursortype (conio.h)
- setfillpattern (graphics.h)

- setfillstyle (graphics.h)
- setgraphbufsize (graphics.h)
- setgraphmode (graphics.h)
- setlinestyle (graphics.h)
- setpalette (graphics.h)
- setrgbpalette (graphics.h)
- settextjunistify (graphics.h)
- settexttyle (graphics.h)
- setusercharsize (graphics.h)
- setviewport (graphics.h)
- setvisualpage (graphics.h)
- setwritemode (graphics.h)
- textheight (graphics.h)
- textwidth (graphics.h)

Интерфейсные подпрограммы

Следующие подпрограммы реализуют обращения к средствам DOS, BIOS и специфическим средствам данного компьютера.

- absread (dos.h)
- abswrite (dos.h)
- bioscom (bios.h)
- _bios_disk (bios.h)
- biosdisk (bios.h)
- _bios_keybrd (bios.h)
- bioskey (bios.h)
- biosprint (dos.h)
- _bios_printer (dos.h)
- _bios_serialcom (dos.h)
- _dos_keep (dos.h)
- _dos_freemem (dos.h)
- freemem (dos.h)

- _harderr (dos.h)
- harderr (dos.h)
- _hardresume (dos.h)
- hardresume (dos.h)
- _hardretn (dos.h)
- hardretn (dos.h)
- keep (dos.h)
- randbrd (dos.h)
- randbwr (dos.h)

Подпрограммы управления памятью

Эти подпрограммы обеспечивают динамическое распределение памяти для моделей данных **small** и **large**.

- allocmem (dos.h)
- _dos_freemem (alloc.h, stdlib.h)
- brk (alloc.h)
- _dos_setblock (dos.h)
- farcoreleft (alloc.h)
- farheapcheck (alloc.h)
- farheapcheckfree (alloc.h)
- coreleft (alloc.h, stdlib.h)
- _dos_allocmem (dos.h)
- farheapchecknode (alloc.h)
- farheapfree (alloc.h)
- farheapwalk (alloc.h)
- farrealloc (alloc.h)
- sbrk (alloc.h)

Разные подпрограммы

Эти подпрограммы предоставляют задержку времени, различные звуковые эффекты и локальные эффекты.

- delay (dos.h)
- sound (dos.h)

- nosound (dos.h)

Глобальные переменные DOS

Ниже приведены глобальные переменные Borland C++, доступные только для 16-разрядных приложений DOS.

_heaplen (dos.h)

Эта переменная содержит длину ближней динамически распределяемой области памяти в малых моделях данных (**tiny**, **small**, **medium**) и описывается следующим образом:

```
extern unsigned _heaplen;
```

В моделях **small** и **medium** размер сегмента данных вычисляется следующим образом:

```
сегмент данных [small,medium] = глобальные данные +
динамически распределяемая область + стек
```

где размер стека можно настроить с помощью **_stklen**.

Если **_heaplen** установлена в 0 (по умолчанию), то программа выделяет для сегмента данных 64К, и размером динамически распределяемой области будет:

```
64К - (глобальные данных + стек)
```

В модели **tiny** все (включая код) находится в одном и том же сегменте, поэтому размер сегмента данных вычисляется следующим образом (с учетом 256 байт для PSP):

```
сегмент данных [tiny] = 256 + глобальные данные +
динамически распределяемая область + стек
```

Если в модели **tiny _heaplen = 0**, то фактический размер динамически распределяемой области вычисляется вычитанием из 64К PSP, кода, глобальных данных и стека.

В моделях **compact** и **large** ближней динамически распределяемой области нет, и стек имеет собственный сегмент, поэтому сегмент данных вычисляется так:

```
сегмент данных [compact, large] = глобальные данные
```

В модели **huge** стек находится в отдельном сегменте, и каждый модуль имеет собственный сегмент данных.

_ovrbuffer (dos.h)

Данная переменная изменяет размер оверлейного буфера и имеет следующий синтаксис:

```
unsigned _ovrbuffer = size;
```

Используемый по умолчанию размер оверлейного буфера равен удвоенному размеру наибольшего оверлея. Для большинства приложений этого достаточно. Однако конкретная функция программы может реализовываться через несколько модулей, каждый из которых является оверлейным. Если общий размер этих модулей больше оверлейного буфера, то при частом вызове модулями друг друга будет происходить дополнительный свопинг.

Решением здесь будет увеличения размера оверлейного буфера, так что в каждый момент времени памяти будет достаточно, чтобы вместить все оверлеи с частыми перекрестными вызовами. Сделать это можно с помощью установки в требуемый размер (в параграфах) глобальной переменной **_ovrbuffer** в 128K:

```
unsigned _ovrbuffer = 0x2000;
```

Для определения оптимального размера оверлейного буфера общего метода не существует.

_stklen (dos.h)

Данная переменная содержит размер стека и имеет следующий синтаксис:

```
extern unsigned _stklen;
```

Переменная **_stklen** определяет размер стека для 6 моделей памяти. Минимально допустимый размер стека — 128 слов. По умолчанию назначается размер 4K.

В моделях данных **small** и **medium** сегмент данных вычисляется следующим образом:

```
сегмент данных [small, medium] = глобальные данных +
динамически распределяемая область + стек
```

где размер динамически распределяемой области можно настроить с помощью **_heaplen**.

В модели **tiny** все (включая код) находится в одном и том же сегменте, поэтому размер сегмента данных вычисляется следующим образом (с учетом 256 байт для PSP):

```
сегмент данных [tiny] = 256 + глобальные данные +
динамически распределяемая область + стек
```

В моделях **compact** и **large** ближней динамически распределяемой области нет, и стек имеет собственный сегмент, поэтому сегмент данных вычисляется так:

```
сегмент данных [compact, large] = глобальные данные
```

В модели **huge** стек находится в отдельном сегменте, и каждый модуль имеет собственный сегмент данных.

Отладчик Turbo Debugger

Назначение отладчика

Турбо отладчик Turbo Debugger представляет собой набор инструментальных средств, позволяющий отлаживать программы на уровне исходного текста и предназначенный для программистов, использующих семейство компиляторов Borland. В пакет отладчика входят набор выполняемых файлов, утилит, справочных текстовых файлов и примеров программ.

Turbo Debugger позволяет вам отлаживать программы для Microsoft Windows, Windows NT и DOS. Многочисленные перекрывающие друг друга окна, а также сочетание спускающихся и раскрывающихся меню обеспечивают быстрый, интерактивный пользовательский интерфейс. Интерактивная, контекстно-зависимая справочная система обеспечит вас подсказкой на всех стадиях работы. Кроме того, Turbo Debugger полный набор средств отладки:

- Вычисление любых выражений языка Си, C++, Pascal и Assemb ler.
- Полное управление выполнением программы, включая программную анимацию.
- Доступ на нижнем уровне к регистрам процессора и системной памяти.
- Полные средства проверки данных.
- Развитые возможности задания точек останова и регистрации.
- Трассировка сообщений Windows, включая точки останова по сообщениям.
- Обратное выполнение.
- Поддержка удаленной отладки, в том числе для Windows.
- Полная поддержка объектно-ориентированного программирования, включая просмотр классов и проверку объектов.

- Макрокоманды в виде последовательности нажатий клавиш, ускоряющие выполнение команд.
- Копирование и вставка между окнами и диалогами.
- Контекстно-зависимые меню.
- Возможность отладки больших программ.
- Диалоговые окна, позволяющие вам настроить параметры отладчика.
- Возможность отладки 16- и 32-разрядных программ Windows (для 32-разрядной отладки имеется отладчик TD32).
- Обработка исключительных ситуаций операционной системы, а также C и C++.
- Сохранение сеанса.
- Поддержка нитей для мультинитевого программирования Windows NT.
- Возможность подключения готовых к выполнению в Windows процессов.
- Возможность выбора для элементов, выводимых в Turbo Debugger, национального порядка сортировки.

Для работы Turbo Debugger требуются те же аппаратные средства, что и для компилятора языка Borland. Кроме того, Turbo Debugger поддерживает графические адаптеры CGA, EGA, VGA, Hercules (монохромный графический режим), Super VGA и TIGA.

Установка и настройка Turbo Debugger

Программа **INSTALL**, поставляемая с компилятором Borland, полностью устанавливает пакет Turbo Debugger, включая выполняемые файлы, файлы конфигурации, утилиты, справочные текстовые файлы и примеры программ. Эта установочная программа создает пиктограммы для компилятора Borland и инструментальных средств языка, помещая их в новую программную группу Windows. Полный перечень файлов, инсталлируемых программой **INSTALL.EXE**, содержится в файле **FILELIST.DOC** (этот файл копируется программой инсталляции в основной каталог компилятора).

Файлы, входящие в состав пакета Turbo Debugger

DUAL8514.DLL

Видео-DLL, которые поддерживают отладку с двумя мониторами.

STB.DLL

Видео-DLL, поддерживающая видео-адаптеры STB.

TD.EXE

Отладчик для отладки приложений DOS.

TDDEBUG.386

Этот драйвер используется TDW.EXE для доступа к специальным отладочным регистрами процессора 80386 (или старше).

TDHELP.TDH

Справочных файл для TD.EXE.

TDKBDW16.DLL

Файл поддержки для Windows.

TDKBDW32.DLL

Файл поддержки для Windows.

TDREMOTE.EXE

Драйвер, используемый в удаленной системе для поддержки удаленной отладки в DOS.

TDVIDW16.DLL

Файл поддержки для Windows.

TDVIDW32.DLL

Файл поддержки для Windows.

TDW.EXE

Выполняемая программа для отладки 16-разрядных программ Windows.

TDW.INI

Файл инициализации, используемый **TDW.EXE**. Он создается программой инсталляции и помещается в основной каталог Windows.

TDWGUI.DLL

Видео-DLL, используемая для вывода окна отладчика TDW в Windows.

TDWHELP.TDH

Справочный файл для TDW.EXE.

TDWINTH.DLL

Поддерживающая DLL для TDW.EXE.

WREMOTE.EXE

Драйвер для удаленной отладки в Windows.

TDINST.EXE

Создает и модифицирует файл конфигурации **TDCONFIG.TD**.

TDMEM.EXE

Выводит на экран доступную память компьютера, включая дополнительную и расширенную.

TDRF.EXE

Утилита передачи файлов, используемая для передачи файлов в удаленную систему.

TDSTRIP.EXE

Удаляет из файлов **.EXE** и **.DLL** используемую отладчиком отладочную информацию (таблицу идентификаторов) без перекомпоновки файлов.

TDUMP.EXE

Выводит на экран структуру 16- или 32-разрядных файлов **.EXE**, **.DLL** и **.OBJ**, а также содержимое отладочную информацию об идентификаторах.

TDWINI.EXE

Позволяет изменить и настроить параметры видеодрайвера отладчика.

TDWINI.HLP

Справочный файл для TDWINI.EXE.

TDWINST.EXE

Создает и модифицирует файл конфигурации.

TDCONFIG.EXE

Настраивает для TDW параметры вывода и цвета экрана.

WRSETUP.EXE

Файл конфигурации для утилиты **WREMOTE** — драйвера удаленной отладки.

TD_ASM.TXT

Файл с информацией по отладке программ на Turbo Assembler, которая полезна также при отладке программ со встроенным ассемблером.

TD_HELP!.TXT

Файл с ответами на общие вопросы. Среди прочего в нем обсуждаются синтаксические отличия между анализом выражений в Turbo Debugger и компиляторах, отладка программ на нескольких языках и др.

TD_HDWMP.TXT

Файл с информации о настройке конфигурации отладчика для использования аппаратных отладочных регистров.

TD_RDME.TXT

Содержит последнюю информацию, отсутствующую в руководстве.

TD_UTILS.TXT

Описывает утилиты отладчика, работающие в режиме командной строки: TDSTRIP, TDUMP, TDWINST, TD32INST, TDSTRP32, TDMEM, TDMAP и TDUMP32.

MAKEFILE

Формирующий файл, используемый в примерах программ.

TDWDEMO.BUG

Исходный код примера программы с ошибками (для отладки).

TDWDEMO.H

Файл заголовка, используемых примером программы.

TDWDEMO.ICO

Пиктограмма примера программы.

TDWDEMO.IDE

Файл проекта для примера программы.

TDWDEMO.RC

Файл ресурса для примера программы.

S_PAINT.C

Исходный код примера программы.

S_PAINT.EXE

Пример программы.

Настройка Turbo Debugger

Вы можете конфигурировать параметры вывода Turbo Debugger и программные установки с помощью файлов конфигурации и меню **Option** отладчика. Параметры, установленные в файлах конфигурации, начинают действовать после загрузке Turbo Debugger. Чтобы изменить параметры после загрузки, используйте команды в меню **Options**.

Файлы конфигурации

При запуске Turbo Debugger использует следующие файлы конфигурации, инициализации и сеанса:

- TDCONFIG.TD
- TFCONFIG.TDW
- TDCONFIG.TD2
- TDW.INI
- XXXX.TR
- XXXXTRW
- XXXX.TR2

Файлы конфигурации TDCONFIG.TD, TDCONFIG.TDW и TDCONFIG.TD2 создаются с помощью отладчиков TD и TDW. Параметры, установленные в этих файлах, переопределяют параметры, используемые данными отладчиками по умолчанию. Вы можете модифицировать файлы конфигурации с помощью инсталляционных программ TDINST.EXE и TDWINST.EXE.

TDW.INI — это файл инициализации, используемый TDW.EXE и TD2.EXE. Он содержит установки для используемого отладчиком видеодрайвера, расположение файла TDWINTH.DLL (динамически компонуемой библиотеки, применяемой для отладчик в Windows), и параметры удаленной отладчик для WRSETUP.EXE.

Программа установки отладчика помещает TDW.INI в основной каталог Windows. В этой копии TDW.INI параметр видеодрайвера устанавливается в SVGA.DLL, а установка DebuggerDLL указывает маршрут к TDWINTH.DLL. Полное описание TDW.INI можно найти в файле TD_HELP!.TXT. Файлы с расширениями **.TR**, **.TRW** и **.TR2** содержат параметры состояния сеанса отладчиков.

Когда вы запускаете Turbo Debugger, он ищет файлы конфигурации в следующей последовательности:

- в текущем каталоге;
- в каталоге, заданном в установке Turbo Directory программы установки Turbo Debugger;
- в каталоге, содержащем выполняемый файл отладчика.

Если Turbo Debugger находит файл конфигурации, то параметры, заданные в этом файле, переопределяют встроенные по умолчанию установки. Если при запуске Turbo Debugger вы указываете параметры командной строки, то они переопределяют соответствующие значения по умолчанию и значения, заданные в файле конфигурации.

Для поддержки доступных типов видеоадаптеров и мониторов TDW использует различные типы видео-DLL. При инсталляции Turbo Debugger запустите программу установки TDWINI.EXE, которая поможет вам выбрать или модифицировать используемые отладчиками видео-DLL. По умолчанию TDW использует драйвер SVGA.DLL, который поддерживает большинство видеоадаптеров и мониторов.

Кроме того, Turbo Debugger поддерживает в TD и TDW отладку с двумя мониторами. Для этого вам потребуется цветной монитор и видеоадаптер и монохромный монитор и видеоадаптер. При отладке с двумя мониторами Turbo Debugger выводится на монохромном мониторе, а отлаживаемая программа — на цветном. Это позволяет видеть во время отладки вывод программы. Для загрузки TD или TDW в режиме с двумя мониторами используйте параметр командной строки **-do**. В файл TDW.INI в раздел **VideoOptions** для этого нужно включить **mono=yes**. Для установки параметров видеоадаптера используйте утилиту TDWINI.EXE.

Меню Options

Язык	**Language...**	**Source**
Макрокоманды	**Macros**	>
Параметры дисплея	**Display options...**	
Маршрут доступа к исходному файлу	**Path for source...**	
Параметры сохранения	**Save options...**	
Параметры восстановления	**Restore options...**	

Это меню содержит команды, с помощью которых вы можете управлять выводом отладчика. С помощью команды **Options Language** вы можете выбрать язык, используемый в Turbo Debugger для вычисления выражений. Команда **Options Dispay Options** открывает диалоговое окно параметров вывода **Display Options**. Параметр этого окна можно использовать для управления выводом Turbo Debugger.

```
[*]               Display options
Display swapping        Integer format
( ) None( )             Hex
(.) Smart               ( ) Decimal
( ) Always              (.) Both
Screen lines            Tab size
(.) 25   ( ) 43/50      8
Background delay User screen delay
10                      3
OK       Cancel         Help
```

Кнопки с зависимой фиксацией **Display Swapping** (Переключение дисплея) позволяет вам выбрать один из трех способов управления переключением между экраном Turbo Debugger и экраном вашей программы, а именно:

- **None** (отсутствует) — нет переключения между экранами. Используйте данный параметр, если вы отлаживаете программу, которая не выводит никакой информации на экран.
- **Smart** (эффективное) — переключение на экран пользователя выполняется только тогда, когда может произойти вывод на экран. Отладчик будет выполнять переключение экранов всякий раз когда вы проходите программу или выполняете инструкцию исходного кода,

в которых осуществляется чтение или запись в видеопамять. Этот параметр используется по умолчанию.

- **Always** (постоянное) — переключение в экран пользователя происходит при каждом выполнении программы пользователя. Используйте этот параметр, если параметр **Smart** не позволяет перехватить все случаи вывода информации на экран вашей программой. Если вы выберете этот параметр, экран будет «мерцать» при каждом шаге выполнения вашей программы, так как на короткое время экран Турбо отладчика будет заменяться на экран вашей программы.

Переключатель Integer Format

Кнопки с зависимой фиксацией Integer Format (Формат целых чисел) позволяет вам определить один из трех форматов, управляющих выводом целых чисел:

- **Decimal** (Десятичный) — целые числа выводятся, как обычные десятичные значения.
- **Hex** (Шестнадцатеричный) — целые числа выводятся в шестнадцатеричном виде в формате, принятом в соответствующем языке.
- **Both** (Оба) — целые числа выводятся и как десятичные, и как шестнадцатеричные значения (которые указываются в скобках после шестнадцатеричных значений).

Переключатель **Screen Lines** (Размер экрана) можно использовать для того, чтобы определить, использует ли Turbo Debugger обычный 25-строчный режим экрана или 40- или 50-строчный режим, доступный при работе с адаптерами EGA и VGA.

Поле **Tab Size** (Размер табуляции) позволяет определить позиции при каждой табуляции. Вы можете уменьшить число позиций табуляции, чтобы можно было видеть больше исходного текста в файлах, выравнивание кода выполнено с помощью табуляции. Размер позиции табуляции можно установить в значения от 1 до 32.

Поле **Background Delay** позволяет вам задать, как часто обновляются экраны отладчика. При использовании этого параметра в сочетании с командой **Run Wait for Child** вы можете

наблюдать действия программы в экранах Turbo Debugger при ее выполнении.

Поле **User Screen Delay** позволяет задать время вывода экрана программы при нажатии **Alt+F5** (команда Windows User Screen). Это полезно использовать при работе в режиме полного экрана, когда вам нужно видеть окна приложения. Определив задержку, вы можете задать, как должно будет выводиться экран программы, прежде чем управление вернется к Turbo Debugger.

Команда **Path for Source** (Маршрут доступа к исходному файлу) задает каталоги, в которых Turbo Debugger будет искать исходные файлы. Чтобы задать несколько каталогов, разделите их точкой с запятой. Хотя поле ввода **Enter Source Directory Path** может содержать максимум 256 символов, для задания более длинных маршрутов вы можете определить файл ответов, содержащий одну строку с определением каталогов. Чтобы задать такой файл в данном поле ввода, введите символ @, затем задайте имя файла.

Команда **Save Options** (Сохранить параметры) открывает диалоговое окно, с помощью которого вы можете сохранить текущие параметры на диске в файле конфигурации. В этом файле сохраняются:

- ваши макрокоманды (кнопка **Options**);
- текущая схема окон и форматы областей окон (**Layout**);
- все значения параметров, заданные в меню **Options** (кнопка **Options**).

```
[*]              Save configuration
[X] Options              OK
[ ] Layout               Cancel
[ ] Macros

Save to                  Help
tdconfig.tdw
```

Поле ввода **Save To** позволяет также задать имя файла конфигурации. По умолчанию TDW.EXE использует TDCONFIG.TDW.

Команда **Restore Options** позволяет восстановить параметры из файла на диске. Вы можете создать несколько файлов конфигурации, записав в них различные макрокоманды, схемы

окон и т.д. Требуется задавать файл параметров, созданный с помощью команды **Options Save Options** или утилиты установки отладчика.

Выполнение программ с отладчиком

Подготовка программ для отладки

Когда вы выполняете компиляцию и компоновку с помощью одного из языков фирмы Borland, вам следует указать компилятору, что нужно генерировать полную информацию для отладки. Если вы скомпилируете объектные модули своей программы без информации для отладки, вам придется перекомпилировать все эти модули, чтобы можно было полностью использовать все средства отладки на уровне исходного кода. Можно также сгенерировать информацию для отладки только для отдельных модулей (это позволит сократить объем программы), но потом будет крайне неприятно попасть в модуль, где информация для отладки недоступна. Поэтому мы рекомендуем перекомпилировать все модули, если, конечно, вам это позволяет имеющаяся память. В случае нехватки памяти или уверенности в правильной работе отдельных модулей можно перекомпилировать только конкретные модули. При компиляции программ для отладки лучше исключить оптимизацию, иначе вы запутаетесь при отладке отдельных частей кода, оптимизированных компилятором.

При компиляции из интегрированной среды для включения в файлы **.OBJ** отладочной информации выберите команду **Options Project** (для вывода **Style Sheet**), в блоке списка **Topic** выберите **Compiler Debugging** и включите в **OBJs** кнопку с независимой фиксацией **Debug**. Чтобы включить отладочную информацию в выполняемые файлы, выберите команду **Options Project**, затем команду **Linker General** в блоке списка **Topic**. Выводятся кнопки с независимой фиксацией **General**. Включите кнопку **Debug Information**.

При компиляции программ с использованием компилятора режима командной строки используйте для включения отладочной информации директиву компилятора **-v**.

После полной отладки программы вы можете скомпилировать и скомпоновать ее заново с оптимизацией и исключением отладочной информации.

Отладка программ ObjectWindows

Если вы применяете TDW для отладки программ, использующих **ObjectWindows**, то нужно конфигурировать отладчик, чтобы он распознавал систему диспетчеризации сообщений **Objecwindows DDVT**. Для этого запустите TDWINST, для вывода диалогового окна **Source Debugging** выберите команду **Options Source Debugging**, включите кнопку с независимой фиксацией **OWL Window Messages**, затем сохраните конфигурацию и выйдите из TDWINST.

Запуск отладчика

После компиляции и компоновки программ с включением отладочной информации вы можете начать процесс отладки, запустив Turbo Debugger и загрузив с ним программу. При этом вы можете использовать один из трех отладчиков:

- **TD.EXE** для отладки 16-разрядных приложений DOS
- **TDW.EXE** для отладки 16-разрядных приложений Windows
- **TD32.EXE** для отладки 32-разрядных приложений Windows.

Отладчики для Windows запускаются в Windows из группу компиляторов Borland в **Program Manager** выбором пиктограмм TDW или TD32, из интегрированной среды компиляторов выбором команды **Tool Turbo Debugger** (программы будут отлаживаться в активном окне **Edit**), из диалогового окна **Program Manager File Run** (в поле ввода **Command** наберите **TDW** или **TD32** и параметры) или из **File Manager** двойным щелчком «мышью» на пиктограмме выполняемого файла TDW.EXE или TD32.EXE из каталога, содержащего Turbo Debugger.

При запуске Turbo Debugger из командной строки можно задать параметры запуска и режимы отладки. Эта командная строка имеет следующий синтаксис:

```
TD   TDW   TD32 [параметры] [имя_программы [аргументы]]
```

Элементы в квадратных скобках не обязательны. При запуске отладчика задавайте корректный маршрут программы и ее аргументы. Параметры Turbo Debugger перечислены ниже:

-ar#

Подключает к процессу с идентификационным номером # и продолжает выполнение.

-as#

Подключает к процессу с идентификационным номером и передает управление Turbo Debugger.

-симя_файла

Файл конфигурации, активизирующийся при загрузке.

-do

Выводит TD.EXE или TDW.EXE на втором дисплее.

-dp

Переключение страниц для TD.EXE.

-ds

Переключение на содержимое экрана пользователя.

-h

Вывод справочного экрана.

-?

Вывод справочного экрана.

-ji

Игнорирование старой информации сохранения.

-jn

Не использовать информацию сохраненного состояния.

-ip

Вывод подсказки, если информация сохраненного состояния старая.

-ju

Использовать информацию сохраненного состояния, даже если она старая.

-k

Разрешает запись нажатий клавиш.

-l

Запуск кода инициализации ассемблера.

-p

Разрешает работать с «мышью».

-r

Отладка на удаленных системах (с параметрами по умолчанию.

-rnлок;удал

Разрешает сетевую отладку.

-rp#

Задает порт для удаленной отладки.

-rs#

Скорость связи: 1 — медленная, 2 — средняя, 3 — быстрая.

-sc

Отмена проверки букв на верхний/нижний регистр.

-sdкат;[кат]

Каталог исходного файла.

-tкаталог

Задает каталог для поиска информации о конфигурации и выполняемых файлов.

-vg

Полное сохранение графики (только для TD.EXE).

-vn

Запрет режима 43/50 строк для TD.EXE.

-vp

Разрешение сохранения палитры EGA/VGA для TD.EXE.

-wc

Разрешает/запрещает сообщение о возможном крахе системы.

-wd

Разрешает проверку на наличие всех DLL вашей программы (по умолчанию разрешена).

Если вы запускаете программу, используя пиктограммы TDW или TD32, то можете задать параметры с помощью диалогового окна **Properties** пиктограммы. При этом параметры сохраняются вместе с установленными значениями характеристик пиктограммы. В окне **Properties** вы можете также задать свою программу и ее аргументы. После этого она будет загружаться при двойном щелчке «мышью» на пиктограмме отладчика. Чтобы задать для пиктограммы значения **Property**, щелкните на ней «мышью», затем выберите в **Program Manager** команду **File Properties**. В поле ввода **Command Line** наберите имя отладчика с параметрами командной строки. После этого щелкните «мышью» на OK.

Для запуска Turbo Debugger из интегрированной среды Borland C++ for Windows, то для задания параметров командной строки можете сделать следующее:

- Для вывода диалогового окна **Tools** выберите команду **Options Tools** интегрированной среды.
- В списке окна **Tools** выберите **TDStartup**.
- Чтобы открыть диалоговое окно **Tools Options**, щелкните «мышью» на командной кнопке **Edit**.
- В поле **Commands Line** после макрокоманды **$TD** введите параметры командной строки отладчика.

Макрокоманда **$ARG** в поле **Command Line** позволяет задать аргументы, передаваемые программе. Чтобы задать аргументы, выберите для открытия диалогового окна **Enviroment Options** команду **Options Enviroment**. Затем выберите в блоке списка **Topics Debugger** и введите в блоке списка **Run Arguments** аргументы программы.

Выполнение отладчика

При выполнении TDW (или TD32) отладчик открывает полноэкранное текстовое окно. Однако, в отличие от других приложений, вы не можете использовать в Turbo Debugger клавиши Windows **Alt+Esc** или **Ctrl+Esc**, то есть смена задач здесь запрещена. Однако в Windows NT TD32 активизирует окно с командной подсказкой, и доступны все обычные средства приложения Windows.

Загрузка программы в отладчик

Программу в Turbo Debugger вы можете загрузить из командной строки или после запуска отладчика. Чтобы загрузить в отладчик новую программу (или сменить загруженную), используйте команду **File Open**. Эта команда открывает набор диалоговых окон, первое из которых называется **Load a Program to Debug**. В TD и TDW это окно содержит дополнительную командную кнопку **Session**, которая используется для поддержки средств удаленной отладки.

В поле ввода **Program Name** задайте имя выполняемого файла программы и нажмите **Enter**. Чтобы выполнить поиск программы по каталогам, щелкните «мышью» на кнопке **Browse**. Откроется второе диалоговое окно — **Enter Program Name to Load**. В блоке **Files** этого окна выводятся файлы в текущем выбранном каталоге. Введя в блоке **File Name** маску файлов (например, ***.EXE**), вы можете задать список нужных файлов.

Для перемещения по каталогам вы можете использовать двойной щелчок «мышью» на записях окна **Directories**. После выбора каталога выберите загружаемый файл в блоке **Files**. Для быстрого поиска файла наберите в блоке **Files** его имя.

После задания программы вы можете определить, требуется ли выполнять в отладчике ее код запуска. Если вы выберите кнопку с независимой фиксацией **Execute Startup Code**, Turbo Debugger выполняет программный код до процедуры **main** программы (или ее эквивалента). В противном случае при загрузке программы никакой код выполняться не будет.

Для поддержки удаленной отладки TDW содержит дополнительный набор переключателей. Если вы выберите в группе **Session** окна **Load a New Program to Debug** кнопку с зависимой фиксацией **Remote**, это позволяет задать отладку на удаленной системе. Кнопка **Local** определяет локальную отладку.

При загрузке программы с включенной в нее отладочной информацией Turbo Debugger открывает окно **CPU**, в котором показывает дизассемблированные инструкции ассемблера. При выполнении программы под управлением отладчика должны быть доступны все ее исходные файлы. Кроме того, в том же каталоге должны находиться все файлы **.EXE** и **.DLL** приложения.

Исходный код программы отладчик ищет в следующем порядке:

- в том каталоге, где компилятор нашел исходные файлы;
- в каталоге, заданном в команде **Options Path for Source** (или в параметре командной строки **-sd**);
- в текущем каталоге;
- в том каталоге, где находятся файлы **.EXE** и **.DLL**.

После загрузки программы в отладчик вы можете с помощью команды **Run Arguments** задать или изменить аргументы программы. Их можно также задать после имени программы в командной строке.

При выходе из Turbo Debugger он сохраняет состояние текущего сеанса в файле сеанса. При перезагрузке программы из этого каталога отладчик восстанавливает параметры последнего сеанса. По умолчанию в файле сеанса сохраняются все списки протоколов, выражения просмотра, элементы буфера, установки исключительных ситуаций операционной системы, установки выражений Си и C++. Эти файлы называются XXXX.TR (отладчик TD), XXXX.TRW (TDW) и XXXX.TR2 (TD32), где **XXXX** — имя отлаживаемой программы. Если при выходе из отладчика программа не загружена, то **XXXX** — это имя отладчика.

Команда **Options Set Restart** открывает диалоговое окно параметров рестарта **Restart Options**, где вы можете настроить обработку в Turbo Debugger файлов сеанса. Кнопка с независимой фиксацией **Restore at Restart** определяет, какие параметры отладчика вы хотите сохранять в файле состояния сеанса, а кнопка с зависимой фиксацией **Use Restart** задает, когда следует загружать файл сеанса:

- **Always** — Файл состояния сеанса используется всегда.
- **Ignore if old—** Если программа перекомпилирована, файл состояния сеанса не используется.
- **Prompt if old** — Turbo Debugger запрашивает, хотите ли вы использовать файл состояния сеанса после изменения программы.
- **Never** — Не использовать файл состояния сеанса.

Управление выполнением программы

В процессе отладки управление периодически передается между вашей программой и отладчиком. Когда управление передается Turbo Debugger, он может использовать свои средства для поиска по исходному коду и структурам данных программы и выявления причины неправильного выполнения программы. Для этого можно использовать меню и окна отладчика. Отладчик предоставляет вам много способов управления выполнением программы. Вы можете:

- выполнять программу по шагам (по одной машинной инструкции или строке исходного кода);
- выполнять как один шаг вызовы функций;
- выполнять программу до заданного места;
- выполнять программу до возврата из текущей функции;
- трассировать программу;
- выполнять программу в обратном направлении;
- выполнять программу до точки останова;
- выполнять программу до появления определенного сообщения Windows;
- приостанавливать программу при возникновении исключительной ситуации C++ или Си.

Кроме точек останова, сообщений Windows и исключительных ситуаций C++ все механизмы управления выполнением находятся в меню **Run**.

Меню Run

Меню **Run** (Выполнение) содержит несколько параметров для выполнения различных частей вашей программы. Поскольку эти параметры часто используются, им соответствуют функциональные клавиши.

Run	**F9**	Выполнение
Go to cursor	**F4**	Выполнение до курсора
Trace into	**F7**	Трассировка
Step over	**F8**	Шаг с пропуском
Execute to…	**Alt-F9**	Выполнение до…
Until return	**Alt-F8**	Выполнение до возврата
Animate…		Автоматизировать…

```
Back trace              Alt-F4      Обратная трассировка
Instruction trace       Alt-F7      Трассировка инструкций
Arguments…                          Аргументы
Program reset           Ctrl-F2     Сброс программы
Next Pending Status                 Следующий ждущий
Wait for Child                      Ожидание дочернего
```

Команда **Run** запускает вашу программу на выполнение. При наступлении одного из следующих событий управление передается отладчику.

- ваша программа завершила выполнение;
- обнаружена точка останова с действием прерывания;
- прервали выполнение с помощью клавиши прерывания;
- выполнение программы остановлено из-за ошибки;
- возникли отмеченные исключительные ситуации Си или C++.

Команда **Go to Cursor** выполняет программу до той строки, где находится курсор (в текущем окне **Module** или области **Code** окна **CPU**). Если текущим окном является окно **Module**, курсор должен находиться на строке исходного кода внутри функции.

Команда **Trace Into** выполняет одну строку исходного кода или машинную инструкцию. Если текущая строка содержит вызов процедуры или функции, то отладчик выполняет трассировку этой процедуры. Однако, если текущим окном является окно **CPU**, то выполняется одна машинная инструкция. Если текущим является окно **Module**, то выполняется строка исходного кода.

Turbo Debugger интерпретирует методы объектов и функции-элементы классов, как все другие процедуры и функции. Клавиша **F7** позволяет трассировать их исходный код (если он доступен).

Если вы выполняете эту команду для одной машинной инструкции, отладчик интерпретирует некоторые инструкции, как одну инструкцию, даже они приводят к выполнению нескольких инструкций. Это инструкции **CALL**, **INT**, **LOOP**, **LOOPZ** и **LOOPNZ**.

Это справедливо и для префиксов **REP**, **REPNZ** или **REPZ**, за которыми следуют инструкции **CMPS**, **CMPSW**, **LODSB**,

MOVS, **MOVSB**, **MOVSW**, **SCAS**, **SCASB**, **SCASW**, **STOS**, **STOSB** или **STOSW**.

Команда **Step Over** выполняет одну строку исходного кода или машинную инструкцию, минуя трассировку вызываемой процедуры или функции. При этом обычно выполняется одна строка исходного текста программы. Если вы используете **Step Over** при расположении указателя на инструкции вызова, то Turbo Debugger полностью отрабатывает эту функции и переводит вас к оператору после вызова функции.

Если вы выполняете эту команду для одной исходной строки, отладчик интерпретирует любой вызов процедуры или функции на этой строке, как часть самой строки, поэтому при завершении вы не окажетесь в начале одной из этих функций. Вместо этого вы перейдете к следующей строке текущей подпрограммы или к предыдущей программе, которая вызвала данную.

Команда **Run Step Over** интерпретирует вызов метода объекта или функцию-элемент класса как один оператор, и выполняет для него такие же действия, как при любом другом вызове процедуры или функции.

Команда **Execute To** выполняет вашу программу до адреса, который вы ввели в ответ на подсказку в диалоговом окне **Enter Code Address to Execute To**. Программа может не достичь этого адреса, если встречается точка останова или вы прерываете выполнение.

Команда **Until Return** выполняет текущую процедуру или функцию, пока она не возвратит управление вызывающей программе. Это полезно использовать при двух обстоятельствах: если вы случайно вошли в процедуру или функцию, выполнение которой вас не интересует (с помощью команды **Run Trace** вместо команды **Run Step**), или когда вы определили, что текущая функция работает правильно, и не хотите медленно проходить по шагам ее оставшуюся часть.

Команда **Animate** выполняет непрерывную последовательность команд **Trace**. Это позволяет вам наблюдать за текущим адресом в исходном коде и видеть изменение значений переменных. Прервать выполнение данной команды можно нажатием любой клавиши. После выбора команды **Run Animate** вам выведется подсказка для ввода значения интервала

временной задержки между последовательными трассировками (в десятых долях секунды). По умолчанию используется значение **3**.

Если вы выполняете трассировку программы (с помощью оперативных клавиш **F7** или **Alt-F7**), то команда **Back Trace** изменяет порядок выполнения на обратный. Это средство удобно использовать, если вы проскочили место предполагаемой ошибки и хотите вернуться к этой точке. Данная команда позволяет вам выполнить программу «в обратном порядке» по шагам или до заданной (подсвеченной) точки в области инструкций окна **Execution History**. В окне **CPU** обратное выполнение доступно всегда, а для исходного кода в окне **Full History** параметр **Execution History** нужно установить в **On**.

Команда **Instruction Trace** выполняет одну инструкцию. Ее можно использовать, когда вы хотите трассировать прерывание, или когда вы находитесь в окне **Module** и хотите выполнять трассировку процедуры или функции, которая находится в модуле без отладочной информации (например, библиотечной подпрограмме). Так как вы больше не будете находиться в начале строки исходного теста, эта команда обычно переводит вас в окно **CPU**.

Команда **Arguments** позволяет вам задать новые аргументы программы. Введите аргументы программы, как они задаются после имени программы в командной строке. После этого отладчик запрашивает, хотите ли вы перезагрузить отладчик с диска. Следует ответить «**Yes**».

Команда **Program Reset** перезагружает отлаживаемую вами программу с диска. Ее можно использовать в следующих случаях:

- когда выполнение «зашло слишком далеко», то есть пройдено то место, где имеется ошибка;
- когда ваша программа завершила работу и вы хотите запустить ее снова;
- если вы работаете в окне **CPU**, приостановили выполнение программы с помощью прерывания и хотите завершить ее и начать сначала (убедитесь однако, что вы не прервали выполнения программы в коде ядра Windows);
- если вы хотите перезагрузить **DLL**, которая уже загружена, установите для нужной **DLL** в **Yes** параметр

Startup Option в диалоговом окне **Load Module Source** или **DLL Symbols** и сбросьте программу.

Если вы выбрали команду **Program Reset** и находитесь в окне **Module** или **CPU**, то отладчик устанавливает **Instruction Pointer** на начало программы, но экран остается там, где вы были при выборе команды **Program Reset**. Такое поведение облегчает установку курсора на то место, где вы были, и выполнение программы до данной строки. Если вы выбрали команду **Program Reset** только потому, что зашли на один оператор дальше нужного места, вы можете переместить курсор в файле исходного кода вверх на несколько строк и нажать клавишу **F4**, чтобы выполнить программу до этого места.

Команда **Next Pending Status** (доступная при отладке в Windows NT) может использоваться при установке в **Yes** команды **Run Wait for Child**. Если **Wait for Child** установлена в **No** (и ваша программа при обращении к отладчику работает в фоновом режиме), то команду **Next Pending Status** можно использовать для получения сообщения о статусе программы. Чтобы указать на наличие такого сообщения, индикатор активности отладчика выводит **PENDING**.

Команду **Wait for Child** (которая используется исключительно отладчиком TD32 для отладки программ Windows NT) можно переключать в **Yes** и **No**. В состоянии **No** вы можете обращаться к отладчику во время выполнения программы, не дожидаясь, пока она дойдет до точки останова. Эта команда полезна при отладке интерактивных программ (она позволяет, например, перейти в отладчик при ожидании программой ввода с клавиатуры).

Прерывание выполнения программы

При выполнении программы вы можете получить доступ к отладчику, нажав клавишу прерывания программы. Используемые клавиши зависят от типа отлаживаемого приложения:

- при отладке программ Windows используйте клавиши **Ctrl+Alt+SysRq**;
- при отладке программ Windows 32s используйте клавиши **Ctrl+Alt+F11**;

- при отладке программ Windows NT используйте клавишу **F12**;
- при отладке программ DOS используйте клавиши **Ctrl+Break**.

Это полезно использовать, когда в программе не установлены точки останова.

Если при возврате в Turbo Debugger вы увидите окно **CPU** без соответствующих программе инструкций, то возможно вы находитесь в коде ядра Windows. При этом следует установить точку останова в том месте, где должна выполняться ваша программа. Затем выполните программу до точки останова (**F9**). После этого можно возобновить отладку. Находясь в ядре Windows, не следует пытаться выполнять программу по шагам или пытаться перезагрузить приложение. Это может привести к краху системы.

Обратное выполнение

Каждую выполненную инструкцию Turbo Debugger регистрирует в протоколе выполнения (при трассировки программы). С помощью окна протокола выполнения **Execution History** вы можете просмотреть выполненные инструкции и вернуться в нужную точку программы. Команда обратного выполнения **Reverse Execute** выполняется по клавишам **Alt+F4**. Turbo Debugger может регистрировать около 400 инструкций. Здесь действуют следующие правила:

- Регистрируются только те инструкции, которые выполнены с помощью команды **Trace Into** (**F7**) или **Instruction Trace** (**Alt+F7**). Однако, если не выполняются отдельные инструкции (перечисленные ниже), то регистрируются также команды **Step Over**.
- Инструкция **INT** приводит к стиранию протокола выполнения. Если вы не трассируете прерывание с помощью **Alt+F7**, то об ратное выполнение этой инструкции невозможно.
- После выполнения команды **Run** или выполнения после прерывания протокол удаляется. (Регистрация начинается после возобновления трассировки.)

- При выполнении вызова функции без ее трассировки обратное выполнение за инструкцию после возврата невозможно.
- Обратное выполнение инструкций работы с портами невозможно (отменить чтение и запись нельзя).
- Невозможно также обратное выполнение вызываемого программой кода Windows (если только вы не находитесь в окне **CPU** и не отлаживаете **DLL**).

В окне **CPU** обратное выполнение доступно всегда, а для обратного выполнения исходного кода нужно установить **Full History** в **On** (в меню **Execution History**). Меню **Execution History** содержит также команды **Inspect** и **Reverse Execute**. Команда **Inspect** переводит вас к команде, подсвеченной в области **Instruction**. Если это строка исходного кода, она выводится в окне **Module**. При отсутствии исходного кода открывается окно **CPU** и подсвечивается инструкция в области **Code**. Действие инструкций **IN**, **INSB**, **INSW**, **OUT**, **OUTSB**, **OUTSW** отменить невозможно, поэтому их обратное выполнение может давать побочные эффекты.

TD.EXE имеет в окне **Execution History** дополнительную область, позволяющую вам вернуться в нужную точку программы при случайной потере протокола. Область **Keystroke Recording** в нижней части этого окна активизируется при разрешении регистрации нажатий клавиш (это можно сделать с помощью **TDINST** или параметра **-k** командной строки).

Область **Keystroke Recording** показывает причину передачи управления отладчику (например, точка останова) и текущий адрес программы с соответствующей строкой исходного кода или машинной инструкцией. Turbo Debugger регистрирует все нажимаемые вами клавиши и записывает их в файл XXXX.TDK, где **XXXX** — это имя отлаживаемой программы. Локальное меню этой области содержит команды **Inspect** и **Keystroke Restore**. По команде **Inspect** отладчик активизирует окно **Model** или **CPU**, в котором курсор позиционирован на ту строку, где нажата клавиша. Команда **Keystroke Restore** перезагружает программу и выполняет ее до строки, подсвеченной в области **Keystroke Recording**.

Завершение программы

При завершении работы программы управление передается в Turbo Debugger, который выводит код выхода программы. После этого любая команда меню **Run** перезагружает программу. После завершения программы проверить или модифицировать ее переменные нельзя.

При выполнении программы в отладчике легко случайно пропустить нужное место. В этом случае вы можете возобновить сеанс отладки с помощью команды **Run Program Reset** (**Ctrl+F2**), которая перезагружает программу с диска. Перезагрузка программы не влияет на точки останова и параметры просмотра.

Выход из отладчика

Завершить сеанс отладки и вернуться в администратор программ Windows вы можете в любое время (за исключением передачи управления в программу или работы с диалоговым окном) с помощью клавиш **Alt+X**. Можно также выбрать команду **File Quit**.

Интерфейс отладчика

Среда Turbo Debugger включает в себя набор меню, диалоговых окон и специальных окон отладчика.

Работа с меню

Команды глобальных меню Turbo Debugger выводятся в верхней части экрана в строке меню. Если вы не находитесь в диалоговом окне, то эти команды всегда доступны. Чтобы открыть меню Turbo Debugger, нажмите **F10**, с помощью стрелок переместитесь в нужному пункту и нажмите **Enter**. После **F10** для перехода к нужному пункту можно также нажать его подсвеченную букву, либо сразу нажмите **Alt+буква** (без **F10**). Системное меню выбирается по **Alt+пробел**. Меню открывается также щелчком «мышью» на соответствующем пункте.

Окна Turbo Debugger

Для вывода информации об отлаживаемой программе в Turbo Debugger используется набор окон. Для облегчения отладки служат команды управления окнами, которые находятся в меню **Window** и **System**. Каждое открываемое окно имеет номер, указанный в его правом верхнем углу. Нажатием клавиши **Alt** в

сочетании с номером окна вы можете активизировать любое из первых 9 окон. Список открытых окон содержится в нижней половине меню **Window**. Чтобы открыть конкретное окно, нажмите в меню **Window** цифру номера окна. Если окон больше 9, в этом меню выводится команда **Window Pick**, выводящая меню окон.

Клавиша **F6** (или команда **Window Next**) позволяет циклически перемещаться по открытым на экране окнам. Окно может иметь несколько областей. Для перемещения между областями используйте клавиши **Tab** или **Shift+Tab**, либо **Window Next**. Курсор в областях перемещается с помощью стандартных клавиш перемещения курсора.

При открытии нового окна оно выводится в месте текущего расположения курсора. Переместить его в другое место можно с помощью команды **Window Size/Move** и клавиш стрелок, либо сразу нажмите **Shift** и сдвигайте окно стрелками. Для быстрого увеличения или уменьшения окна выберите **Window Zoom** (**F5**) или щелкните «мышью» на кнопке минимизации/максимизации в верхнем правом углу окна.

Если вы по ошибке закрыли окно, вернуться в последнее окно можно с помощью команды **Window Undo Close** (**Alt+F6**). Когда программа затирает своим выводом экран операционной среды (при выключенном переключении экрана), вы можете очистить его с помощью **System Repaint Desktop**. Для возврата к используемой по умолчанию схемы окон Turbo Debugger выберите **System Restore Standard**.

Каждое окно Turbo Debugger имеет специальное оперативное меню **SpeedMenu**, содержащее команды, относящиеся к данному окну. Области окон также могут иметь свои меню. Для доступа к **SpeedMenu** активного окна или области вы можете нажать в окне правую кнопку «мыши», либо нажать клавиши **Alt+F10**, либо нажать **Ctrl** и подсвеченную букву команды **SpeedMenu** (для этого должно быть разрешено действие команд-сокращений).

Окна меню View

Меню **View** является точкой входа в большинство окон Turbo Debugger. Перечислим их кратко. С помощью команды **View**

Another вы можете дублировать на экране окна **Dump**, **File** и **Module**.

Окно Breakpoints

Используется для установки, модификации или удаления точек останова. Точка останова определяет то место в программе, где отладчик приостанавливает выполнение программы. Это окно имеет две области. Справа перечислены условия и действия точек останова, слева — все точки останова.

Окно Stack

Показывает текущее состояние программного стека. Первая вызванная функция показывается в нижней части окна, а выше ее — каждая последующая. Подсвечивая эти функции и нажимая **Ctrl+I** вы можете проверять исходный код. Кроме того, можно открыть окно **Variables** и вывести все локальные переменные и аргументы функции (**Ctrl+L**).

Окно Log

Выводит содержимое журнала сообщений с прокручиваемым списком сообщений и информацией, сгенерированной при работе с отладчиком. Это окно можно также использовать для получения информации об использовании памяти, модулях и оконных сообщения приложения Windows.

Окно Watches

Показывает значения переменных и выражений. Введя в это окно выражения, вы можете отслеживать их значения при выполнении программы. Окно добавляется с помощью клавиш **Ctrl+W** при установке курсора на переменной в окне **Module**.

Окно Variables

Выводит все переменные в данном контексте программы. В верхней области окна перечисляются глобальные переменные, а в нижней — локальные. Это полезно использовать для поиска функции или идентификатора, имени которых вы точно не помните.

Окно Module

Одно из важнейших окон Turbo Debugger, показывающее исходный код отлаживаемого программного модуля (включая **DLL**). Модуль должен компилироваться с отладочной информацией.

Окно File

Выводит содержимое любого файла на диске. В нем можно просматривать шестнадцатеричные байты или текст ASCII и искать нужные байтовые последовательности.

Окно CPU

Выводит текущее состояние процессора. Окно имеет 6 областей, где выводятся дизассемблированные инструкции, селекторы Windows (только в TDW), шестнадцатеричные данные, стек в шестнадцатеричном виде, регистры ЦП и флаги процессора. Это окно полезно использовать при отладке программ на ассемблере или просмотре точно последовательности инструкций.

Окно Dump

Выводит в шестнадцатеричном виде содержимое любой области памяти (аналогично области окна **CPU**). Команды **SpeedMenu** этого окна позволяют вам модифицировать данные и работать с блоками памяти.

Окно Registers

Показывает содержимое регистров (в области регистров) и флагов ЦП (в области флагов). С помощью команд **SpeedMenu** вы можете изменить их значения.

Окно Numeric Processor

Показывает текущее состояние сопроцессора и имеет три области: содержимого регистров с плавающей точкой, значений флагов состояния и значений управляющего флага. Это позволяет вам диагностировать проблемы в использующих сопроцессор подпрограммах.

Окно Execution History

Выводит последние выполненные машинные инструкции или исходные строки программы, номер строки исходного кода и

следующую выполняемую инструкцию или строку кода. Используется для обратного выполнения.

Окно Hierarchy

Выводит на экран дерево иерархии всех используемых текущим модулем классов. Имеет область списка классов и дерева иерархии. Это окно показывает взаимосвязь используемых в модуле классов.

Окно Windows Messages

Показывает список оконных сообщений программы Windows. Области этого окна показывают задание режима отслеживания сообщений, тип перехватываемых сообщений и перехваченные сообщения.

Окно Clipboard

Буфер Clipboard отладчика используется для для вырезания и вставки элементов из одного окна отладчика в другое. Оно показывает вырезанные элементы и их типы. Скопированные в буфер элементы динамически обновляются.

Окна Inspector

Выводят текущее содержимое выбранной переменной. Его можно открыть с помощью команды **Data Inspect** или **Inspect** меню **SpeedMenu**. Закрывается оно обычно по **Esc** или щелчком «мышью» на блоке закрытия. При последовательном открытии нескольких окон **Inspector** нажатием **Alt+F3** или командой **Window Close** вы можете закрыть сразу все эти окна. Окна **Inspector** выводят простые скалярные величины, указатели, массивы, объединения, структуры, классы и объекты. Выбором команды **Inspect** в этом окне вы можете создать дополнительные окна **Inspector**.

Экран пользователя

Экран пользователя показывает полный экран вывода вашей программы. Этот экран имеет такой же вид, как при выполнении программы без Turbo Debugger. Чтобы переключиться в этот экран, выберите команду **Window User Screen**. Для возврата в экран отладчика нажмите любую клавишу.

Специальные средства Turbo Debugger

Автоматическое дополнение имени

Когда в поле ввода выводится подсказка для ввода имени идентификатора, вы можете набрать часть имени, а затем нажать **Ctrl+N**. Turbo Debugger заполнит остальную часть имени автоматически. При этом набранная часть должна уникальным образом идентифицировать имя. Если с набранных символов не начинается ни одно из имен, то ничего не происходит. При наличии нескольких идентификаторов, соответствующих набранным вам символам, выводится список имен, из которого вы можете выбрать нужное.

Выбор по набору

Некоторые окна позволяют вам начать набор нового значения, не выбирая сначала команду **SpeedMenu**. Выбор по набору обычно применяется к наиболее часто используемым командам **SpeedMenu**.

Инкрементальное сопоставление

Это средство помогает вам находить записи в алфавитных списках. При наборе каждого символа полоса подсветки перемещается к первому элементу, начинающемуся с выбранных вами букв. Позиция курсора указывает, какую часть имени вы уже набрали. После подсветки вы можете нажать **Alt+F10** или щелкнуть правой кнопкой «мыши». При этом выводится **SpeedMenu**, где вы можете выбрать команду, соответствующую подсвеченному элементу.

Клавиатурные макрокоманды

Макрокоманды представляют собой просто определяемые вами оперативные клавиши. Одной клавише вы можете назначить любую последовательность команд и нажатий клавиш.

Расположенная в меню **Options** команда **Macros** выводит всплывающее меню с командами для определения клавиатурных макрокоманд и удаления ненужных: **Create** (**Alt+=**), **Stop Recording** (**Alt+-**), **Remove** и **Delete All**. Команда **Create** начинает запись макрокоманды, а команда **Stop Recording** завершает ее (не используйте для завершения записи команду **Options Macro Stop**

Recording, так как она добавится к вашей макрокоманде). **Delete All** удаляет все текущие макрокоманды.

Работа с буфером Clipboard

Чтобы скопировать элемент в буфер Clipboard, позиционируйте на элементе курсор, нажмите клавишу **Ins** для его подсветки, затем нажмите клавиши **Shift+F3**. Чтобы вставить содержимое буфера в окно или диалоговое окно, нажмите **Shift+F4**. Выводится диалоговое окно **Pick**, содержащее список всех элементов буфера Clipboard и набор кнопок с зависимой фиксацией, позволяющих вам выполнять различным образом вставку элементов: **String**, **Location** и **Contents**. Это позволяет вам интерпретировать элемент, как вставляемый одним из трех способов: как строку, как адрес, или как содержимое по адресу. Категории, которые вы можете использовать для вставки элемента, зависят от его типа и назначения.

Для вставки элемента в диалоговое окно, подсветите элемент, выделите соответствующую категорию, затем нажмите клавишу **Enter** или активизируйте кнопку **OK** (для редактирования записи) или **Paste** (если вы хотите отредактировать запись).

Выбор команды View Clipboard выводит на экран окно Clipboard, в котором перечисляются все вырезанные элементы.

```
[*] Clipboard
Module          :     @#TCDEMO#36 nlines
Inspector       :     nlines                     0 (0x0)
Module          :     @#TCDEMO#38 totalcharacters
Inspector       :     totalcharacters            0 (0x0)
```

В левом поле этого окна описывается тип записи, за которым следует двоеточие и вырезанный элемент. Если вырезанный элемент представляет собой выражение из окна **Watch**, переменную из окна **Inspector** или данные, регистр или флаг из окна **CPU**, то за элементом следует его значение или значения.

Address

Адрес без соответствующих данных или кода.

Control flag

Значение управляющего флага сопроцессора.

Coprocessor

Регистр арифметического сопроцессора.

CPU code

Адрес и список байт выполняемых инструкций из области кода окна **CPU**.

CPU data

Адрес и список байт данных в памяти из области данных в окне **CPU** или в окне **Dump**.

CPU flag

Значение флага ЦП из области флагов окна **CPU**.

CPU register

Имя регистра и значение из области регистров окна **CPU** или окна **Register**.

CPU stack

Исходная позиция и кадр стека из области стека окна **CPU**.

Expression

Выражение из окна **Watches**.

File

Позиция в файле (в окне **File**), которая не является модулем в программе.

Inspector

Одно из следующих:

- имя переменной из окна **Inspector**;
- значение константы из окна **Inspector** или **Watch**;
- регистровая переменная окна **Inspector**;
- битовое поле окна **Inspector**.

Module

Содержимое модуля, включая позицию в исходном коде, аналогично переменной из окна **Module**.

Status flag

Значение флага состояния сопроцессора.

String

Текстовая строка, например, отмеченный блок из окна **File**.

При вставке элементов из буфера Clipboard их тип должен соответствовать типу поля ввода. **SpeedMenu** окна Clipboard содержит следующие команды:

Inspect

Позиционирует курсор в то окно, из которого был извлечен элемент.

Remove

Удаляет подсвеченный элемент или элементы. Тот же эффект для подсвеченного элемента имеет клавиша Del.

Delete all

Удаляет все в буфере Clipboard.

Freeze

Приостанавливает динамическое обновление элемента Clipboard.

Текстовое окно Get Info

Вы можете выбрать команду **File Get Info** для анализа использования памяти и определения того, почему получил управление отладчик. Эта и другая информация отображается в текстовом блоке, который удаляется с экрана при нажатии клавиши **Enter**, пробела или **Esc**. В этом окне отображается следующая информация, в зависимости от того, отлаживаетесь ли вы в DOS, или в Windows.

Если вы отлаживаете программу для DOS, то в блоке **System Information** будет выведена следующая информация:

- имя отлаживаемой вами программы;
- описание причины остановки программы;
- объемы памяти, используемой DOS, отладчиком и вашей программой;
- версия DOS или Windows, под управлением которой вы работаете;
- текущая дата и время.

TDW дает вам следующую информацию о глобальной памяти:

Mode (режим)

Режимами памяти могут быть:

- **Large-frame EMS** (EMS-память с большим размером страничного блока)
- **Small-frame EMS** (EMS-память с малым размером страничного блока)
- **non-EMS** (дополнительная память).

Banked (банкируемая)

Объем памяти в килобайтах выше линии банка EMS (которая может быть откачана в расширенную память, если ее использует система).

Not Banked (не банкируемая)

Объем памяти в килобайтах ниже линии банка EMS (которая не может быть откачана в расширенную память).

Largest (наибольший)

Наибольший непрерывный блок памяти в килобайтах.

Symbols (идентификаторы)

Объем оперативной памяти, используемый для загрузки таблицы идентификаторов программы.

Кроме перечисленной выше информации окно **Windows System Information** для Windows NT содержит также следующую информацию:

- **Memory Load Factor** (процент используемой оперативной памяти)
- **Physical** (доступный и общий объем системной памяти)
- **Page file** (размер текущего страничного файла и максимальный размер)
- **Virtual** (общая и доступная виртуальная память).

Команда Attach

Эта команда позволяет подключить TD32 к процессу, работающему под Windows NT. Ее полезно использовать, когда вы знаете, где в программе возникают ошибки, но вам трудно

воспроизвести ситуацию в отладчике. Команда открывает диалоговое окно **Attach to and Debug a Running Process**. Для подключения к выполняемому процессу сделайте следующее:

- Запустите процесс, который нужно отладить.
- Запустите TD32.
- Выберите команду **File Change Dir** для перехода в каталог выполняющегося процесса.
- Выберите команду **File Attach** для открытия диалогового окна.
- Включите или выключите кнопку с независимой фиксацией **Stop an Attach**. При ее включении Turbo Debugger приостанавливает выполнение процесса при подключении к нему.
- В блоке списка **Processes** выберите процесс (или введите идентификационный номер процесса в поле ввода **Process ID**). Затем щелкните «мышью» на **OK**.

Если процесс содержит информацию об отладке, и Turbo Debugger может найти исходный код, открывается окно **Module**. В противном случае открывается окно **CPU**. После этого вы можете использовать отладчик и отлаживать процесс как обычно.

Команда OS Shell

Эта команда отладчика TD32 работает в операционной системе Windows NT. По этой команде Turbo Debugger открывает командную подсказку. Для возврата в отладчик наберите **Exit**.

Получение справочной информации

Turbo Debugger предлагает несколько способов получения в ходе отладки справочной информации. С помощью клавиши **F1** вы можете получить доступ к развитой контекстной справочной системе. По данной клавише на экран выводится список тем, из которых вы можете выбрать необходимую.

Индикатор активности в левом правом углу экрана всегда показывает текущее состояние. Например, если курсор находится в окне, в индикаторе активности выводится **READY**. Если выводится меню, в нем указывается **MENU**, а если вы находитесь в диалоговом окне — **PROMPT**. Если вы, запутаетесь и не можете

понять, что происходит в отладчике, взгляните на индикатор активности.

В строке состояния в нижней части экрана всегда дается краткая информация об используемых клавиатурных командах. При нажатии клавиши **Alt** или **Ctrl** данная строка изменяется. Когда вы находитесь в системе меню, эта строка предлагает вам оперативное описание текущей команды меню.

Оперативная помощь

В отладчик встроен контекстно-зависимый оперативный справочник. Он доступен как при работе в системе меню, так и при выводе сообщения об ошибке или подсказки. Для вывода справочного экрана с информацией, относящийся к текущему контексту (окну или меню) нажмите клавишу **F1**. При наличие «мыши» вы можете вывести справочный экран, выбрав **F1** в строке состояния. Некоторые справочные экраны содержат подсвеченные слова, которые позволяют вам получить дополнительную информацию по данной теме. Для перемещения к нужным ключевым словам используйте клавиши **Tab** или **Shift+Tab** и нажмите клавишу **Enter**. Для перемещения к первому или последнему слову на экране используйте клавиши **Home** и **End**. Доступ к оперативным справочным средствам можно получить также с помощью команды **Help** из строки меню (оперативные клавиши **Alt+H**).

Если вы хотите вернуться к предыдущему справочному экрану, нажмите клавиши **Alt+F1** или выберите команду **Previous** из меню **Help**. В справочной системе для просмотра последних 20 экранов можно пользоваться клавишей **PgUp** (клавиша **PgDn** работает, когда вы находитесь в группе связанных экранов). Для доступа к индексному указателю справочной системы нажмите **Shift+F1** (или **F1** в справочной системе) или выберите команду **Index** в меню **Help**. Для получения информации о самой справочной системе выберите в меню **Help** команду **Help Help**. Для выхода из справочной системы нажмите клавишу **Esc**.

При работе в отладчике в нижней части экрана выводится краткая справочная строка. В этой строке состояния кратко описаны клавиши или команды меню для текущего контекста.

Точки останова

В Turbo Debugger понятие точки останова включает в себя три следующих элемента:

- место в программе (адрес), где находится точка останова;
- условие, при котором она срабатывает;
- что происходит, когда срабатывает точка останова (действие).

Адрес может представлять собой отдельный адрес в программе или быть глобальным (при этом останов может происходить на любой строке исходного кода или инструкции программы). Под условиями могут подразумеваться следующие условия, когда происходит останов:

- всегда;
- когда выражение принимает истинное значение;
- когда объекты данных изменяют свое значение.

Можно также задавать «счетчик проходов», который определяет, чтобы прежде чем сработает точка останова, «условие» должно принимать истинное значение определенное число раз.

При достижении точки останова может выполняться следующее действие:

- приостановка выполнения программы;
- регистрация значения выражения;
- выполнение выражения;
- разрешение группы точек останова;
- запрещение группы точек останова.

Обычно точка останова устанавливается на конкретной исходной строке или машинной инструкции программы. Когда программа достигает точки останова, Turbo Debugger вычисляет ее. Однако точки останова могут быть и глобальными. Глобальные точки останова вычисляются отладчиком после выполнения каждой строки исходного кода или инструкции. Это позволяет определить момент модификации переменной или указателя.

Когда программа доходит до точки останова, Turbo Debugger проверяет условия точки останова и проверяет истинность заданного условия. Если условие выполняется, точка останова срабатывает. Такая точка останова называется условной.

Окно Breakpoints

Создать окно точек останова **Breakpoints** можно с помощью команды **View Breakpoints** основного меню. Это дает вам способ выбора и установки условий, при которых срабатывает точка останова. Это окно можно использовать для добавления новых точек останова, отмены (удаления) точек останова и изменения существующих точек останова.

```
[*] Breakpoints
TCDEMO.220    Breakpoint
TCDEMO.225    Always
TCDEMO.226    Enabled
```

В левой области этого окна показан список всех адресов, где установлены точки останова. В правой области показаны подробные данные по текущим (подсвеченным в левой области) точкам останова.

Локальное меню **SpeedMenu** окна **Breakpoints** можно получить по нажатию клавиш **Alt+F10**. Команды данного меню позволяют вам добавлять новые точки останова, отменять существующие или изменять характер поведения имеющихся точек останова.

Установка простых точек останова

Когда вы впервые устанавливаете точку останова, Turbo Debugger создает по умолчанию простую точку останова. При достижении такой точки останова программа всегда приостанавливает выполнение. Чтобы выполнить программу до точки останова, нажмите **F9**.

Простейшие методы установки простых точек останова предлагают окно **Module** и область **Code** окна **CPU**.

Если вы работаете с клавиатурой, поместите курсор на любую выполняемую строку исходного кода или инструкцию в области кода окна **CPU** и нажмите **F2**. То же самое можно сделать с помощью команды **Breakpoint Toggle**. После установки точки

останова соответствующая строка становится красной. Для отмены точки останова нажмите **F2**.

При работе с «мышью» вы можете установить точку останова, щелкнув на двух левых столбцах нужной строки. Повторный щелчок «мышью» отменяет точку останова.

Кроме того, команда **Breakpoinr At** (**Alt+F2**) позволяет установить простую точку останова на текущей строке. Кроме того, эта команда открывает диалоговое окно **Breakpoint Options**, которое предоставляет быстрый доступ к командам настройки точки останова.

Кроме установки точке останова из окон **Module** и **CPU**, Turbo Debugger предлагает для установки точек останова следующие команды. Чтобы установить простые точки останова на точках входа во все функции текущего загруженного модуля или все функции-элементы класса, используйте команду **Group** локального меню окна **Breakpoints**. Команда **Add** этого же меню также устанавливает точки останова. Она открывает диалоговое окно **Breakpoint Options** и позиционирует курсор на пустое поле ввода **Address**, где вы можете ввести адрес или номер строки.

После установки точки останова вы можете модифицировать действие, выполняемое по ее активизации. По умолчанию это «**Break**» — Turbo Debugger приостанавливает выполнение программы.

Установка условных точек останова

Эти точки останова также устанавливаются по конкретному адресу в программе, однако имеют специальные условия и связанные и ними действия.

Иногда точку останова нежелательно активизировать при каждом ее обнаружении, особенно когда содержащая ее строка выполняется многократно. Не всегда также желательно приостанавливать программу на точке останова. В таких случаях используются условные точки останова. Для создания условной точки останова можно выполнить следующие шаги:

- Установите простую точку останова (как описано выше).
- Откройте диалоговое окно **Conditions and Actions**.
  - Откройте окно точке останова и подсветите в области **List** нужную точку останова.

- Выберите в **SpeedMenu** команду **Set Options**. Выводится диалоговое окно **Breakpoint Options**. Это окно содержит команды, позволяющие модифицировать параметры точек останова. Текущие параметры выбранной точки останова выводятся в блоке списка **Conditions and Actions&**
- Чтобы модифицировать условие точки останова и выполняемые по ней действия, щелкните «мышью» на командной кнопке **Change**. Выводимое окно **Conditions and Actions** позволяет вам настроить условия срабатывания точки останова и выполняемые по ней действия.

- Выберите кнопку с зависимой фиксацией **Expression True**. По умолчанию условие точек останова устанавливается в **Always**, то есть они срабатывают каждый раз при обнаружении их в программе. Щелчок «мышью» на кнопке с зависимой фиксацией **Expression True** задает активизацию точки останова только после того, как заданное вами выражение станет истинным.
- В поле ввода **Condition Expression** введите выражение. Оно будет вычисляться при каждом обнаружении точки останова.
- Если нужно, задайте для точки останова счетчик проходов **Pass Count**. Это поле определяет, сколько раз должно удовлетворяться условие точки останова, прежде чем точка останова будет активизирована. По умолчанию он равен 1. Значение счетчика уменьшается при каждом удовлетворении условия.
- Если вы хотите изменить выполняемое по умолчанию в точке останова действие, щелкните «мышью» на нужно кнопке с зависимой фиксацией группы **Action**.
- Для выхода из окна щелкните «мышью» на **OK** или нажмите **Esc**.

Установка точек останова по изменению памяти

Эти точки останова отслеживают выражения, при вычислении которых получается объект памяти или адрес. Они активизируются при изменении значения соответствующего объекта данных или указателя памяти. Для установки такой точки

останова нужно выполните те же шаги, что и перечисленные выше, но:

- В диалоговом окне **Conditions and Actions** вместо **Expression True** щелкните «мышью» на кнопке с зависимой фиксацией **Changed Memory**.
- В поле ввода **Condition True** введите выражение, при вычислении которого получается объект памяти или адрес.

Когда ваша программа обнаруживает строку с такой точкой останова, условное выражение вычисляется перед выполнением этой строки. Это нужно учитывать.

При вводе выражения вы можете также ввести счетчик числа отслеживаемых объектов. Общее число отслеживаемых байт памяти равно произведению размеру объекта, на которое ссылается выражение, на счетчик объекта.

Установка глобальных точек останова

Эти точки останова являются по существую точками останова двух описанных выше типов, но отслеживаются они непрерывно в течении всего периода выполнения программы. Так как Turbo Debugger проверяет такие точки останова после выполнения каждой инструкции или строки исходного кода, они являются превосходным инструментом выявления того места в программе, где происходит порча данных.

Чтобы создать глобальную точку останова, установите сначала условную точку останова или точку останова по изменению памяти (как описано выше), затем после выхода из окна **Conditions and Actions** включите кнопку с зависимой фиксацией **Global** диалогового окна **Breakpoint Options**.

Поскольку глобальные точки останова не связываются с конкретными адресами программы, в поле ввода **Address** диалогового окна **Breakpoint Options** выводится <**not available**>.

Чтобы глобальная точка останова проверялась после выполнения каждой машинной инструкции, а не каждой строки исходного кода, в активном окне **CPU** нажмите **F9**. Эти точки останова сильно замедляют выполнение программы, поэтому использовать их нужно умеренно. Кроме того, для них не рекомендуется задавать условие «**Always**».

Меню **Breakpoint** содержит команды для быстрой установки глобальных точек останова: **Changed Memory Global** и **Expression True Global**. При этом по умолчанию выбирается действие «**Break**». **Changed Memory Global** устанавливает глобальную точку останова, активизируемую при изменении значения в памяти. Эта команда выводит подсказку для задания соответствующей области памяти **Enter Memory Address** и поле счетчика **Count**. **Expression True Global** устанавливает точку останова, срабатывающую при истинном значении заданного выражения.

Аппаратные точки останова

Эти точки останова доступны в TDW и TD32 при отладке программ Windows NT. Они используют специальные отладочные регистры процессоров Intel 80386 и старше. Эти точки останова являются глобальными. Для работы с этими точками останова вам потребуется драйвер TDDEBUG.386. Скопируйте его с дистрибутивных диске и включите в файл CONFIG.SYS. (Инструкции содержатся в файле TD_HDWBP.TXT.) При правильной установке этого драйвера в поле **Breakpoints** диалогового окна **File Get Info** выводится **Hardware** (в противном случае — **Software**).

Чтобы установить аппаратную точку останова, выберите в меню **Breakpoints** команду **Hardware Breakpoint**. Эта команда автоматически устанавливает кнопку **Global** окна **Breakpoint Options**, кнопку **Hardware** в окне **Conditions and Actions** и открывает диалоговое окно **Hardware Breakpoint Options**. Это окно содержит все параметры аппаратных точек останова и полностью описано в файле TD_HDWBP.TXT.

Можно также создать аппаратную точку останова, модифицировав существующую точку останова:

- Установите кнопку с независимой фиксацией **Global** в диалоговом окне **Options**.
- Откройте диалоговое окно **Conditions and Actions** и выберите кнопку с зависимой фиксацией **Hardware**.
- Чтобы открыть диалоговое окно **Hardware Breakpoint Options**, щелкните «мышью» на кнопке **Hardware** окна **Conditions and Actions**.
- Задайте параметры аппаратной точки останова и щелкните «мышью» на **OK**.

- Если нужно, задайте в окне **Conditions and Actions** нужные действия.

Действия, выполняемые по точкам останова

Кнопка с зависимой фиксацией **Action** в диалоговом окне **Conditions and Actions** позволяет задать действия, выполняемые по точке останова.

Break

Break приводит к тому, что при срабатывании точки останова программа останавливается. Экран отладчика будет выведен заново, и вы можете вводить команды для просмотра структур данных программы.

Execute

Execute приводит к выполнению выражения (выражение запрашивается в поле ввода **Action Expression**). Выражение должно иметь некоторые побочные эффекты, например, присваивание значения переменной. Эта возможность позволяет вам включить выражение, которое будет выполняться перед кодом вашей программы в строке с текущим номером («вставка кода»). Такое средство полезно использовать, когда вы хотите изменить поведение подпрограммы, чтобы проверить «диагноз» или скорректировать ошибку. Это позволяет при проверке минимальных изменений в программе не выполнять цикл компиляции и компоновки.

Log

Log приводит к тому, что значение выражения будет записано в окне **Log**. Вам выводится подсказка. В ответ на нее вы должны ввести выражение, значение которого требуется зарегистрировать. Будьте внимательны, чтобы выражение не имело никаких неожиданных побочных эффектов.

Enable group

Enable group позволяет вновь активизировать запрещенную ранее группу точек останова. Укажите в поле ввода **Action Expression** номер группы.

Disable group

Disable group позволяет запретить группу точек останова. При запрещении группы точек останова они не стираются, а

просто маскируются на время сеанса отладки. Укажите в поле ввода **Action Expression** номер группы.

Задание условий и действий

Для задания активизации точки останова и того, что должно при этом происходить, используется окно **Conditions and Actions**. Обычно для каждой конкретной точки останова задается одно условие или выражение действия. Однако отладчик позволяет задавать несколько выражений. Кроме того, с одной точкой останова можно связать несколько условий и действий.

Чтобы задать набор условий, выберите кнопку с зависимой фиксацией **Changed Memory of Expression**, введите в поле ввода **Condition Expression** условие, выберите кнопку **Add** под блоком ввода **Condition Expression** (если вводится несколько выражений, повторите эти шаги). Кнопка **Delete** под полем **Condition Expression** позволяет удалить из поля ввода **Condition Expression** текущее подсвеченное выражение.

При выборе кнопки с зависимой фиксацией **Execute**, **Log**, **Enable Group** или **Disable Group** в группе **Action**, нужно задать набор условий, по которым Turbo Debugger будет активизировать точку останова. Набор условий состоит из одного или более выражений. Чтобы задать их, выберите кнопку с зависимой фиксацией **Execute**, **Enable Group** или **Disable Group**, введите действие в поле ввода **Action Expression** и выберите кнопку **Add** под полем ввода **Action Expression**. Чтобы при активизации точки останова выполнять более одного выражения, повторите эти шаги. При задании нескольких условий и действий они вычисляются в порядке их ввода.

При выборе кнопки **Enable Group** или **Disable Group** для ссылки на группы точек останова, которые нужно разрешить или запретить, наберите в поле **Action Expression** номер группы.

Кнопка **Delete** под полем **Action Expression** позволяет удалить из набора действие текущее подсвеченное выражение. Закончив ввод действий, выберите в диалоговом окне **Condition Action** командную кнопку **OK**.

Условия и действия точки останова управляются заданными выражениями. Turbo Debugger вычисляет выражение точки останова относительно области действия того места, где находится точка останова. Используя синтаксис переопределения

области действия, вы можете обращаться к значениям любого определенного объекта данных, однако это замедляет вычисления.

Чтобы модифицировать точку останова в другом (не загруженном в данный момент) модуле, используйте команду **View Another Module**.

Группы точек останова

Объединение точек останова в группы позволяет разрешать, запрещать или удалять их одним действие. Кроме того, одной командой можно задать группу точек останова для всех функций (или функций-элементов) модуля.

Команда **Group** в локальном меню окна **Breakpoint** активизирует диалоговое окно **Edit Breakpoint Groups**, с помощью которого вы можете создать или модифицировать точки останова.

```
[*]             Edit Breakpoint groups
Group
1         #BCDEMO#38 #BCDEMO#39             OK
3         #BCDEMO#40

                                            Help

Add...   Delete   Enable   Disable
```

Группа точек останова идентифицируется положительным целым числом, которое автоматически генерируется отладчиком или назначается вами. Отладчик автоматически присваивает групповое число каждой создаваемой точке останова. Генерируемый номер группы представляет собой наименьший еще не использованный номер. Таким образом, если номера 1, 2 и 5 уже используются группами, то следующей создаваемой точке останова автоматически присваивается номер группы 3. После создания точки останова вы можете модифицировать статус ее группы с помощью команды **Breakpoint Groups**.

Кнопка **Add** окна **Edit Breakpoints** активизирует диалоговое окно **Add Group**, содержащее один блок списка и набор кнопок с зависимой фиксацией. Блок списка **Module/Class** выводит список модуле или классов текущей программы. Посветите нужных модуль или класс и выберите **OK**. Все устанавливаемые таким образом точки останова объединяются в одну группу. Кнопка

Delete удаляет подсвеченную группу, а **Enable/Disable** разрешают или временно запрещают данную группу.

Кнопки с зависимой фиксацией позволяют выбрать тип функций, выводимых в блоке **Module/Class**: кнопка **Modules** выбирает все модули в текущей программе, а кнопка **Classes** — все ее классы.

Удаление точек останова

Удалить точки останова можно с помощью локального меню (**SpeedMenu**) окна **Breakpoints** или меню **Breakpoints**. Команда **Remove** меню окна **Breakpoints** или клавиша **Del** стирают точку останова, подсвеченную в области списка. Команда **Delete All** меню **Breakpoint** и локального меню окна **Breakpoints** удаляют все установленные точки останова.

Точки останова в шаблонах C++

Turbo Debugger поддерживает размещение точек останова в шаблонах C++, шаблонах функций и шаблонах экземпляров классов и объектов. Для установки таких точек останова используются следующие методы:

- Если точка останова устанавливается нажатием **F2** в окне **Module**, то точки останова задаются для всех экземпляров классов в шаблонах. Это позволяет вам отладить поведение шаблона.
- Если для установки точки останова в шаблоне используются клавиши **Alt+F2**, то активизируется диалоговое окно **Breakpoint Options**, и в поле ввода **Address** вы можете задать адрес шаблона. Открываемое диалоговое окно позволяет вам выбрать конкретный экземпляр класса.
- Установить точку останова на конкретном экземпляре класса шаблона можно также с помощью окна **CPU**. Позиционируйте курсор на строке кода шаблона и нажмите **F2**.

Удаляются такие точки останова аналогично другим: позиционируйте курсор на точке останова в окне **Module** и нажмите **F2**. Удаляются все точки останова соответствующих экземпляров классов. Конкретные точки останова можно удалить с помощью окна **CPU**.

Установка точек останова в нитях

Программы для Windows NT состоят из одной или более выполняемых «нитей». При их отладке вы можете установить точки останова в конкретных нитях, даже если этот код совместно используется несколькими нитями. По умолчанию точка останова в программе Windows NT устанавливается во всех нитях программы. Чтобы установить ее только в одной нити, сделайте следующее:

- Подсветите нужную точку останова в области списка окна **Breakpoint**.
- Выберите команду локального меню **Set Options**.
- Чтобы открыть диалоговое окно **Conditions and Actions**, щелкните «мышью» в на кнопке **Change** диалогового окна **Breakpoint** Options. Если нужно, установите для точки останова условия и действия. По умолчанию отмечается кнопка **All Threads** — точки останова устанавливаются во всех активных нитях.
  - Сбросьте установку **All Threads**. Становится доступным поле ввода **Threads**. Наберите в этом поле номер нити Windows NT. (Чтобы получить номер нити Windows NT, с помощью команды **View Thread** откройте диалоговое окно **Thread**. В области **Threads List** выводятся все активные нити.)
  - Чтобы подтвердить установку, выберите командную кнопку **OK**.

Окно Log

Это окно отслеживает события, происходящие во время сеанса отладки. Открывается оно по команде **View Log** и по умолчанию содержит до 50 строк текста (вы можете изменить это с помощью программы инсталляции).

```
[*] Log                              3
At MCINPUT.124
Breakpoint at TCDEMO.220
Breakpoint at TCDEMO.220
Breakpoint at TCDEMO.220
We are now entering procedure Params...
Breakpoint at TCDEMO.180
```

В это окно записываются:

- адрес программы при ее приостановке;
- комментарий (при использовании команды **Add Comment** данного окна);
- значение выражения, определенного для активизированной точки останова;
- содержимое области или окна при выборе команды **Edit Dump to Log**;
- информация о локальной и глобальной динамической распределяемой памяти или список программных модулей (при выборе команды **Display Windows Info** локального меню данного окна);
- при установке в **Yes** параметра **Send to Log Window** окна **Windows Messages** в окно **Log** передаются все посылаемые данному окну сообщения.

Команды **SpeedMenu** окна **Log** позволяют вам записывать журнал в файл на диске, останавливать и начинать регистрацию, комментировать журнал, очищать его и записывать в него информацию о программе Windows.

Open Log File

Эта команда записывает на диск все записи, регистрируемые в окне **Log**. Вам выводится подсказка для ввода имени файла на диске. По умолчанию он имеет расширение **.LOG**, а его имя соответствует имени программы. При открытии файла в него записываются все уже зарегистрированные записи. Если это нежелательно, выберите сначала команду **Erase Log**.

Close Log File

Закрывает файл, открытый с помощью команды **Open Log File**.

Logging

Разрешает/запрещает запись событий в окно **Log**. Используется для управления регистрацией событий.

Add Comment

Позволяет включить в окно **Log** комментарии. Открывает диалоговое окно с подсказкой для ввода комментария.

Erase Log

Очищает окно **Log**. Файл журнала на диске не изменяется.

Display Windows Info

Доступна только для TDW и выводит на экран окно **Windows Information**. Позволяет вывести информацию о динамически распределяемой памяти и список модуля приложения.

Анализ и модификация данных

Данные вашей программы — это глобальные и локальные переменные, а также определенные константы. Для проверки и модификации данных в Turbo Debugger имеется ряд окон.

Окно Watches

Это окно обеспечивает самый простой способ отслеживания элементов данных программы. В нем вы можете просматривать переменные и выражения, значения которых нужно отслеживать.

```
[*]             Watches                         2
wordcount unsigned int 8 (0x8)
wordcounts      unsigned int [10] {1,2,4,6,1,1,2,0,0,0}
lettersinfo struct linfo [26]
{(4,2),(1,1),(0,0),(1,1),(7,0),(.
nlines*nwords   unsigned int 24 (0x22)
totalcharacters  unsigned long 88L (0x42)
```

Это окно допускает просмотр значений как простых переменных, так и составных объектов данных (например, массивов). Элементы составных объектов выводятся в фигурных скобках (**{}**). Можно также отслеживать выражения, не ссылающиеся непосредственно на память. Отслеживаемые выражения перечисляются в левой части окна, соответствующие типы данных и значения — справа.

Чтобы задать отслеживаемые данные, выберите команду **Data Add Watch**, либо команду **Watch** локального меню окна **Module**, **Variable** или **Watches**. Turbo Debugger открывает диалоговое окно **Enter Expression to Watch**. Введите в нем имя переменной или выражение.

Если в окне **Module** курсор находится на переменной, то она автоматические добавляется в окно **Watch** при выборе окна

Wathes в **SpeedMenu**. Это же относится к выражениям, выделенными с помощью клавиш **Ins** и стрелок.

Если не переопределяется область действия, отладчик вычисляет выражения относительно текущего указателя команд. Если выражение содержит символ, недоступный в активной области действия, то выводятся символы **????**. При вводе выражений вы можете использовать имена еще не определенных переменных, поэтому имена следует вводить аккуратно (Turbo Debugger не распознает ошибок).

При трассировке внутри функции-элемента можно использовать указатель **this**, который можно сопровождать спецификаторами формата и квантификаторами.

Меню окна Watches

SpeedMenu окна **Wathes** содержит все команды, необходимые для работы с элементами окна.

Wathes

Эта команда выводит подсказку для ввода имени переменной или выражения, добавляемого в окно **Watches**. Если не задается область действия, оно вычисляется относительно текущей позиции курсора.

Edit

Открывает диалоговое окно **Edit Watch Expression**, позволяющее вам модифицировать подсвеченное в окне **Wathes** выражение.

Remove

Удаляет из окна **Watches** подсвеченный элемент.

Delete All

Удаляет из окна **Watches** все выражения. Ее полезно использовать при при перемещении из одной области программы в другую.

Inspect

Открывает окно **Inspector** с детальной информацией по подсвеченному в окне **Watch** элементу. Ее полезно применять для просмотра сложного объекта данных.

Change

Модифицирует значение текущей подсвеченной в окне **Wathes** переменной. При вводе в диалоговом окне **Enter New Value** нового значения Turbo Debugger выполняет необходимое преобразование типа.

Окно Variables

В этом окне, которое открывается по команде **View Variable**, показаны все локальные и глобальные переменные (с именами и значениями), доступные из текущего места программы. Его можно использовать, чтобы найти переменные, написание имен которых вы не помните. Для дальнейшего анализа или изменения их значений можно использовать команды локальных меню. Это окно можно также использовать для проверки переменных, локальных по отношению к любой вызванной функции.

```
[*]             Variables                       3
TCDEMO.SHORESULTS           @7129:01fA
TCDEMO.INIT                 @7129:0402
TCDEMO.PROCESSLINE          @7129:04B5
TCDEMO.PARMSONHEAP          @7129:0651
TCDEMO.NUMLINES             1 ($1)
TCDEMO.NUMWORDS             0 ($0)

CH                          'A
ISLETTER                    True
S                           'ABC DEF'
I                           1 ($1)
WORDLEN                     28969
```

Окно имеет две области. Область глобальных переменных (вверху), показывает все глобальные идентификаторы программы. Область статических/локальных переменных (внизу) показывает все статические переменные (идентификаторы) текущего модуля. В обеих областях выводится имя переменной (слева) и ее значение (справа). Если отладчик не может найти информации о типе данных идентификаторов, то он выводит четыре вопросительных знака (**????**).

Меню окна Variables

Каждая область окна **Variables** имеет собственное **SpeedMenu**. Оба меню содержат команды **Inspect**, **Change** и **Wathes**, а команда **Show** имеется только в области локальных идентификаторов.

Inspect

Открывает окно **Inspector**, где выводится содержимое подсвеченного идентификатора. В отличие от обычных окон **Inspector**, если вы проверяете глобальную переменную, имя которой совпадает с именем локальной переменной, то Turbo Debugger выводит значение глобальной переменной. При проверке имени функции активизируется окно **Module**, а курсор перемещается на имя этой функции в исходном коде (при его отсутствии выводится окно **CPU**).

Change

Открывает диалоговое окно **Change**, в котором можно изменит значение подсвеченного идентификатора.

Watch

Открывает окно **Watches** и добавляет в него подсвеченный идентификатор. При этом не отслеживается, глобальная это переменная или локальная. В блоке локальной переменной локальная переменная имеет старшинство.

Show

Выводит диалоговое окно **Local Display**. Кнопки с зависимой фиксацией этого окна позволяют разрешить или изменить область действия переменной в области локальных переменных.

- **Show** — показывать только статические переменные.
- Auto — только переменные, локальные для текущего блока.
- **Both** — и статические, и локальные (по умолчанию).
- **Module —** смена текущего модуля. Выводит диалоговое окно со списком модулей программы.

Переменные стека

С помощью окна **Stack** вы можете проверить любые переменные или функции, которые находятся в стеке (включая рекурсию). Для этого откройте окно стека и подсветите

проверяемую функцию. Затем нажмите **F10** и выберите **Locals**. Область **Statis** окна **Variables** показывает значения аргументов.

Окна Inspector

Эти окна предоставляют наилучший способ просмотра элементов данных, так как они автоматически форматируются в соответствии с типом данных. Их особенно полезно использовать при проверке сложных объектов данных (массивов или связанных списков). Чтобы просмотреть данные в шестнадцатеричном виде, в активном окне **Inpsector** используйте команду **View Dump**. Окна **Inspector** открываются из команды **Data Inspector** или **SpeedMenu** окон **Wathes**, **Variables** или **Inspector**.

При открытии окна **Inspector** выводится диалоговое окно **Enter Variable** с подсказкой на ввод выражений. Введите имя переменной или выражение. Если в момент команды **Inspect** курсор находится на идентификаторе, или вы выделили выражение, то они автоматически помещаются в поле ввода. Заголовок окна **Inspector** содержит проверяемое выражение.

Скалярное окно **Inspector** показывает значения простых элементов данных, таких как **char**, **int** или **long**. Оно содержит две строки: в первой указан адрес переменной, а вторая показывает ее тип и значение (в десятичном/шестнадцатеричном виде).

```
[*] Inspecting wordcount                3
05A51:AA00
unsigned int          2 (0x02)
```

Окно **Inspector** для указателей выводит значения переменных, указывающих на другие элементы данных. В верхней строке указывается адрес переменной, а далее следует детальная информация об указываемых данных. В нижней области показывается тип этих данных.

```
[*] Inspecting bufp                     3
register  ds:0874 [TCDEMO buffer]
[0]                                     'n' 110 (0x88)
[1]                                     'O' 111 (0x6F)
[2]                                     'w' 119 (0x77)

char *
```

Если указатель ссылается на сложный объект данных, значения заключаются в фигурные скобки (выводится столько данных, сколько можно показать). При ссылке на строку символов выводится каждый элемент символьного массива с указанием индексов и значений. Команда **Range** позволяет выводить несколько строк информации.

Окна **Inspector** для структур и объединений показывают значения элементов в сложных объектах данных. Такое окно имеет две области. В верхней области выводится адрес объекта данных с перечислением имен и значений элементов данных объекта. Нижняя область содержит одну строку. Если вы в верхней области подсветите адрес объекта данных, в нижней выводится тип объекта и его имя. В противном случае там показывается тип элемента данных, подсвеченного в верхней области.

```
[*] Inspecting letterinfo[n]                    3
$7937:0852
count           2 (0x2)
firstletter          2 (0x2)

struct linfo
```

Область **Inspector** для массива показывает значения элементов массива (каждому элементу соответствует строка). Слева выводится индекс, справа — значение. Если значением является составной объект, Turbo Debugger выводит максимум данных объекта.

```
[*] Inspecting letterinfo         3
$7682:0852
[0]                               {2,2}
[1]                               {2,0}
[2]                               {2,0}
[3]                               {1,1}
[4]                               {1,0}

struct linfo [26]
```

Окно **Inspector** для функции показывает адрес функции, ее аргументы, а также возвращаемый функцией тип (в нижней области) и соглашения по вызову.

```
[*] Inspecting analyzewords             3
071E9:02DD
char *bufp

long ()
```

Меню окон Inspector

SpeedMenu окон **Inspector** содержит ряд полезных команд.

Range

Задает начальный элемент и число элементов, которые нужно просмотреть в массиве.

Change

Позволяет изменить значение подсвеченного элемента на значение в окне **Enter New Value**. Необходимое приведение типа выполняется автоматически.

Inspect

Открывает новое окно **Inspector** с элементом, подсвеченным в текущем окне **Inspector**. Используется для проверки составных объектов данных. Эту команду можно вызвать, подсветив элемент и нажав **Enter**. Если текущий элемент является функцией, то выводится окно **Module**. Для возврата в прежнее окно нажмите **Esc**. Чтобы закрыть все окна **Inspector**, дайте команду **Window Close** (**Alt+F3**).

Descend

Эта команда работает аналогично команде **Inspect** локального меню, но она заменяет окно **Inspector** и выводит новые элементы. Это позволяет уменьшит число выводимых окон **Inspector**. Однако при использовании **Descend** для структуры данных вы не сможете вернуться к предыдущему просмотру.

New Expression

Позволяет вам проверить другое выражение, которое замещает данные в текущем окне **Inspector**.

Окно Stack

Это окно позволяет проанализировать стек вызова и вывести в удобном для чтения формате все активные функции и значения аргументов. Окно **Stack** вы можете создать с помощью

команды **View Stack**. В окне стека выводится список всех активных процедур и функций. Первой в списке указывается последняя вызванная процедуры, за которой следует вызвавшая ее процедура и предыдущая процедура, и так до самой первой функции программы (функция **main** в Си). Это окно выводит также имена функций-элементов, перед которой указывается имя класса. При рекурсивном вызове окно **Stack** содержит несколько экземпляров функции.

```
[*]  Stack                                  3
TCDEMO.PROCESSLINE.ISLETTER('A')
TCDEMO.PROCESSLINE('ABCDEF')
```

SpeedMenu окна **Stack** содержит две команды: **Ispect** и **Locals**. Команда **Inspect** открывает окно **Module** и позиционирует курсор на активную строку подсвеченной функции. Если подсвеченная функция находится в вершине стека вызова (последняя вызванная функция), то в окне **Module** показывается положение счетчика команд. В противном случае курсор позиционируется на строку после вызова соответствующей функции. Вызвать эту команду можно также нажатием **Enter** после подстветки нужной функции. Команда **Locals** открывает окно **Variables** с идентификаторами, локальными для текущего модуля и подсвеченной функции.

Команда Evaluate/Modify

Эта команда меню **Data** открывает диалоговое, которое содержит текст по текущей позиции курсора или выражение, выбранное с помощью **Ins** и стрелок, затем вычисляет его (если вы выберите кнопку **Eval**) так же, как это сделал бы компилятор. Результат помещается в поле **Result**.

```
[*] Evaluate/Modify
Expression                                  Eval
thisShape[CurrentPoint]
CurrentShape == LINE
HIWORD<lParam>
                                            Cancel
Result
struct SSHAPE <<113,116,0,0>,5,1,0,0>       Help
New value                                   Modify
<not avaliable>
```

Диалоговое окно содержит три поля:

- В поле ввода **Expression** вы можете ввести выражение для вычисления. После содержит протокол всех введенных выражений.
- В средней области выводится результат вычисления вашего выражения. Если строки данных слишком велики и не умещаются в поле результата, то они заканчиваются символом >. «Прокрутив» окно вправо, вы можете просмотреть остаток строки.
- Нижняя область **New Value** — это область ввода, в которой вы можете ввести новое выражение для вычисления. Если выражение модифицировать нельзя, то в данной области выводится сообщение <**not avaliable**>.

Запись в поле ввода **New Value** (Новое значение) будет действовать, если вы выберите кнопку **Modify**. Если вы выполняете отладку объектно-ориентированных программ С++, то окно **Evaluate/Modify** позволяет вам также вывести поля объекта или элементы экземпляра класса. Для каждого элемента, который может использоваться при вычислении записи, можно использовать спецификатор формата.

Команда Function Returns

По команде **Function Returns** выводится возвращаемое текущей функцией значение. Используйте эту команду только тогда, когда функция собирается передать значение в вызывающую программу. Возвращаемое значение выводится в окне **Inspector** (Проверка), поэтому вы легко можете просмотреть значения, представляющие собой указатели на сложные объекты данных. Данная команда позволяет вам не переходить в окно **CPU**, когда требуется просмотреть возвращаемое через регистры процессора значение.

Вычисление выражений

Выражение — это последовательность идентификаторов программы, констант и операций языка, при вычислении которого получается значение. Оно должно соответствовать синтаксису и правилам выбранного языка.

Механизм вычисления выражений Turbo Debugger

При вводе выражения в отладчике оно передается механизму вычисления выражения, который проверяет синтаксис и вычисляет значения идентификаторов, возвращая вычисленное значение. Чтобы задать механизм вычисления выражений, выберите команду **Options Language**, которая открывает диалоговое окно **Expression Language**. Это окно содержит кнопки с зависимой фиксацией **Source**, **C**, **Pascal** и **Assembler**, задающие язык для вычисления выражений. Кнопка **Source** определяет вычисления в соответствие с исходным языком. В большинстве случаев Turbo Debugger поддерживает полный синтаксис указанных языков.

Типы выражений

Вы можете использовать выражения для доступа к значением идентификаторов программы, вычисления значений и изменения значений элементов данных. Допускается задавать шестнадцатеричные значения, адреса памяти, строки программы, байтовые списки и вызовы функций. Формат записи шестнадцатеричного значения зависит от выбранного механизма вычисления:

Язык	16-разрядный	32-разрядный
C	0xnnnn	0xnnnnnnnn
Pascal	$nnnn	$nnnnnnnn
Assembler	0nnnn	0nnnnnnnn

При отладке 16-битового кода для задания адреса памяти вы можете использовать обозначение «сегмент:смещение», например:

Язык	Формат	Пример
C	Oxnnnn	Ox1234:Ox0010
Pascal	$nnnn	$1234:0010
Assembler	nnnnh	1234h:0B234h

Чтобы задать номер строки программы, перед десятичным номером строки укажите символ #. Можно задавать также байтовые списки:

Язык	Список	Данные
C	1234"AB"	34 12 41 42
Pascal	"ab"0x04"c"	61 62 04 63
Assembler	'ab'$04'c'	61 62 04 63

Функции из выражений вызываются также, как в исходном коде. Это позволяет быстро проверить поведение функции.

Выражения с побочными эффектами

Побочный эффект означает изменение при вычислении выражения элемента данных. Это мощный инструмент отладки. Побочные эффекты имеют выражения с операциями присваивания (=, += и др.) и выражения с операциями ++ и --.

Спецификаторы формата

Чтобы изменить используемый по умолчанию формат вывода, укажите после выражение запятую и один из спецификаторов:

c

Символ или строка выводятся на экран в виде необработанных символов. Обычно непечатаемые символы выводятся в виде управляющих символов или в числовом формате. Этот параметр приводит к тому, что при выводе символов будет использоваться полный набор символов дисплея IBM.

d

Целое число выводится в виде десятичного значения.

f[#]

Формат с плавающей точкой с заданным числом цифр. Если вы не задаете число цифр, то используется столько цифр, сколько необходимо.

m

Выражение со ссылкой на память выводится в виде шестнадцатеричных байт.

md

Выражение со ссылкой на память выводится в виде десятичных байт.

P

Выводится необработанное значение указателя, показывающее сегмент, как имя регистра (если это возможно). Показывается также объект, на который указатель ссылается. Если управление форматом не задано, то это используется по умолчанию.

s

Выводится массив или указатель на массив символов (строка, заключенная в кавычки). Строка завершается нулевым символом.

x или h

Целое выводится в виде шестнадцатеричного значения.

Если спецификатор формата не применим к типу данных выражения, он игнорируется. Вы можете задать таким же образом счетчик повторения (он указывает, что выражение относится к повторяющемуся элементу данных, например, массиву).

Переопределение области действия

Область действия идентификатора — это та область программы, в которой на него можно ссылаться. Заданные в выражении идентификаторы Turbo Debugger ищет в следующем порядке:

- идентификаторы в стеке текущей функции;
- идентификаторы в модуле, содержащем текущую функцию;
- глобальные идентификаторы (вся программа);
- глобальные идентификаторы в DLL, начиная с первой загруженной DLL.

Для определения области действия идентификатора отладчик использует текущую позицию курсора. Если вы измените в отладчике область действия, это может дать непредсказуемые результаты, поэтому для возврата к текущей точке используйте команду **Origin** окна **Module**.

Синтаксис переопределения области действия зависит от выбранного в окне **Options Language** языка. В Си, C++ и ассемблере для этого используется символ #, в Pascal — точка. Таким образом, для переопределения области действия

используется следующий синтаксис (в квадратные скобки заключены необязательные элементы):

```
[#модуль[#имя_файла]]#номер_строки[#номер_переменной]
```

или

```
[#модуль[#имя_файла]][#имя_функции]#имя_переменной
```

Просмотр и модификация файлов

Turbo Debugger предусматривает два способа просмотра файлов на диске: окно **Module** и окно **File**. Окно **Module** чаще всего используется в отладчике. Его можно применять для просмотра исходного кода выполняемого модуля, скомпилированного с отладочной информацией. Строка заголовка этого окна показывает имя текущего загруженного модуля, имя текущего исходного файла и номер строки курсора. Выполняемая строка в этом окне помечается символом точки (.), а стрелка (>) в первой позиции показывает указатель команд. Он всегда отмечает следующий выполняемый оператор. При загрузке программы в отладчик окно **Module** загружается автоматически

При выполнении программы по шагам окно **Module** автоматически показывает исходный код, соответствующий выполняемой инструкции. Перемещаясь по исходному коду, вы можете установить точки останова и задать отслеживаемые выражения, а также проверить значения переменных. Если в строке заголовка выводится **opt**, то программа оптимизирована компилятором. Это может затруднить поиск переменных. Если файл модифицирован после последней компиляции, то в заголовке выводится **modified**. Это может привести к несоответствию строк исходного текста. Перекомпилируйте программу.

Команды меню окна Module

Меню **SpeedMenu** окна **Module** содержит команды, позволяющие перемещаться по исходному тексту, выбирать и просматривать элементы данных и загружать новые исходные файлы. В TD32 это меню содержит дополнительные команды **Thread** и **Edit**.

Inspect

Открывает окно **Inspector** с подробной информацией о переменной программы в позиции курсора (если курсор не установлен на переменной, выводится подсказка). Для быстрого

перемещения и выбора выражений в окне **Module** используйте стрелки и клавишу **Ins**. После выбора выражения активизируйте окно **Inspector** с помощью **Ctrl+I**.

Watch

Добавляет переменную в текущей позиции курсора в окно **Watch**. Включение переменной в окно **Watches** позволяет отслеживать ее значение при выполнении.

Thread

Открывает диалоговое окно **Pick a Thread**, из которого вы можете выбрать для отслеживания конкретную нить программы.

Module

Команда **Module (F3)** позволяет выбрать в диалоговом окне **Load Module Source or DLL** и загрузить в отладчик другой модуль.

File

Позволяет просмотреть другой исходный файл, входящий в состав данного модуля. Открывает диалоговое окно **Pick a Source File** с перечнем исходных файлов, содержащихся в выполняемом коде. При выборе нового файла он заменяет в окне **Module** текущий. Чтобы просматривать их одновременно, используйте команду **View Another Module**.

Previous

Возвращает вас к тому месту исходного кода, которое вы просматривали перед сменой позиции.

Line

Позиционирует вас на новую строку с указанным номером, который задается в выводимом диалоговом окне **Enter New Line Number**.

Search

Ищет заданную строку символов, начиная с текущей позиции курсора. Строка задается в выводимом диалоговом окне **Enter Search String**. Если курсор позиционирован на имени переменной, то окно инициализируется этим именем. Чтобы инициализировать окно **Search String**, вы можете также выделить с помощью **Ins** и стрелок блок файла. В строке поиска можно задавать трафаретные символы * и ?.

Next

Ищет следующий экземпляр заданной в команде **Search** строки.

Origin

Позиционирует курсор на модули и строку, соответствующую текущей инструкции. Ее полезно использовать для возврата в исходное место.

Goto

Открывает окно **Enter Address to Position To**, в котором можете ввести любой адрес программы, который хотите просмотреть (в виде имени процедуры или в шестнадцатеричном виде). Это окно выводится также при наборе в окне **Module**.

Edit

При отладке программ Windows с помощью TD32 с помощью этой команды вы можете вызвать выбранный редактор. Это полезно использовать для коррекции исходного кода перед выходом из отладчика. Вызов редактора требует настройки конфигурации с помощью TDINST32.EXE (команда **Options Directories**).

Exceptions

Если вы реализовали на Си или C++ обработку исключительных ситуаций, то доступна эта команда.

Просмотр других файлов

Для просмотра любого файла на диске, включая двоичные и текстовые, используйте окно **File**. При выборе в строке меню команды **View File** отладчик выводит диалоговое окно **Enter Name of File**. Вы можете задать в нем трафаретные символы или конкретное имя файла. В зависимости от содержимого файла в открываемом окне **File** файлы выводятся в текстовом или шестнадцатеричном виде.

Команды окна File

Команды **SpeedMenu** окна **File** можно использовать для перемещения по файлу и изменения формата вывода.

Goto

Позиционирует вывод на новую строку (при просмотре текстового файла) или смещение в файле (при шестнадцатеричном выводе).

Search

Ищет строку символов, начиная с текущей позиции курсора. Для ввода строки выводится окно **Enter Search String**. При шестнадцатеричном выводе можно задать список байт (в соответствии с используемым языком). Допускаются трафаретные символы (* и ?).

Next

Ищет следующий экземпляр строки, заданной в команде поиска.

Display As

Переключает вывод между текстовым и шестнадцатеричным форматом.

File

Позволяет сменить файл, выводимый в окне **File**. Окно **File** не дублируется. Чтобы просматривать два файла одновременно, выберите команду **View Another File**.

Edit

Эквивалентна соответствующей команде окна **Module**.

Отладка на уровне ассемблера

При отладке программы на языке высокого уровня обычно достаточно отладки на уровне исходного кода. Однако иногда может потребоваться проанализировать программу глубже.

Окно CPU

Это окно открывается командой View CPU строки меню и использует различные области для описания состояния вашей программы на нижнем уровне. Его можно использовать для:

- просмотра машинного кода и дизассемблированных инструкций программы;
- проверки и модификации байт структур данных программы;

- тестирования исправления ошибок с помощью встроенного ассемблера в области кода.

```
                                   область регистров
                    область кода          область стека

[*] CPU 80486                                3    [^][ ]
TCDEMO.120: Inc(NumLines);               ^ ax 0004   c=0
  cs:04C4:4F36063000 inc word ptr [TPDEMO  bx 3EEE   z=0
TCDEMO.121  i := 1;                        cx 0000   s=0
 cs:04C8 C:43FE0100  word ptr [bp+02].000  dx 5920   o=0
TCDEMO.122: while i <= Length(S) do        si 3CEC   p=0
 cs:04C0 C47ED4    les  di,[bp+04]         bp 3EF4   a=0
 cs:0400 288A05    mov  al,es:[di]         sp 3EF4   i=1
 cs:0403 3D84      xor  ah,ah              ds 5920   d=0
 cs:0405 3B48FE    cmp ax,[bp+02]          es 5920
 cs:0408 7D03      jnl TPDEMO.125 (04DD)   ss 595A
 cs:040A 898A00    jmp TPDEMO.148          cs 548A
TCDEMO.125 while (i <= Length(S)) and notv ip 04C8
<                                        >
 ds:0008 5A 5D 5A 5D 5A 5D 00 00  Э^$< <   ss:3EF2 548A
 ds:0010 00 00 00 00 00 00 5A 5D  6D  vЖ   ss:3EF0>04C1
 ds:0018 00 00 5A 5D 00 00 00 90    7      ss:3EEE 0246
                область дампа            область стека
```

Область кода показывает машинный код и дизассемблированные машинные инструкции вашей программы. Здесь могут также выводиться строки исходного кода. В области регистров выводится содержимое регистров ЦП. В области флагов показывается состояние 8 флагов процессора. В области дампа выводится шестнадцатеричный дамп любой области памяти, доступной для программы. Область стека показывает шестнадцатеричное содержимое стека программы. Область селекторов доступна только для TDW и показывает все селекторы Windows.

Для адресных ссылок вне текущего сегмента в окне **CPU** выводятся знаки вопроса. Клавиша **Ctrl** в сочетании со стрелками позволяет сдвигать вывод на 1 байт. При выполнении кода Windows, модуля без отладочной информации, остановке программы на инструкции внутри строки исходного кода или при трассировке инструкций с помощью **Alt+F7** окно **CPU** выводится автоматически.

Область кода

В левой части области кода выводятся адреса дизассемблированных инструкций. Для 16-разрядного кода они

имеют вид «**сегмент:смещение**», а для 32-разрядного это 32-разрядные адреса. Стрелка (>) справа от адреса памяти указывает текущий адрес программы (следующую выполняемую инструкцию). Справа выводится шестнадцатеричный машинный код с соответствующей дизассемблированной инструкцией. Глобальные идентификаторы выводятся в виде имени, статические — в виде имени модуля с символов # и именем идентификатора, а номера строк представлены как имя модуля, # и номер строки. Клавиша **F2** позволяет устанавливать/отменять точки останова.

Меню **SpeedMenu** области кода содержит команды, позволяющие перемещаться по ней и ассемблировать вводимые инструкции. TDW имеет дополнительную команду ввода-вывода, а TD 32 — команды **Threads** и **OS Exceptions**.

Goto

Вам выводит окно **Enter Address to Position To** для ввода нового адреса, на который вы хотите перейти. Вы можете ввести адрес, выходящий за пределы программы, что позволяет проверить базовую систему ввода-вывода (BIOS), внутренние области DOS и Windows.

Origin

Позиционирует вас на текущий адрес программы. Используется для перемещения.

Follow

Позиционирует область кода по целевому адресу текущей подсвеченной инструкции. Используется в сочетании с инструкциями передачи управления (**CALL**, **JMP**, **INT**) и условного перехода (**JZ**, **JNE**, **LOOP** и др.).

Caller

Позиционирует вас на инструкцию, вызвавшую текущее прерывание или подпрограмму. Если текущая подпрограмма прерывания занесла данные в стек, то Turbo Debugger может не иметь возможности определить, откуда она вызвана.

Previous

Восстанавливает позицию области кода в соответствии с адресом, который был текущим перед последней командой, явно изменившей его значение. Использование клавиш перемещения на команду не влияет.

Search

Позволяет вам вводить инструкцию или список байт, которые вы хотите найти. Будьте внимательны при поиске инструкций. Следует выполнять поиск только тех инструкций, которые не изменяют байт, в которые они ассемблируются, в зависимости от того, где в памяти они ассемблируются. Например, поиск следующих инструкций проблемы не представляет:

```
PUSH     DX
POP      [DI+4]
ADD      AX,100
```

а попытка поиска следующих инструкций может привести к непредсказуемым результатам:

```
JE       123
CALL     MYFUNC
LOOP     $-10
```

Вместо инструкции можно вводить также список байт.

View Source

Для вывода исходного кода, соответствующего текущей дизассемблированной инструкции открывает окно **Module**. Если соответствующего исходного кода нет (например, вы находитесь в коде **Windows**, или отсутствует отладочная информация), вы просто остаетесь в области кода.

Mixed

Позволяет выбрать один из трех способов вывода на экран дизассемблированных инструкций и исходного кода:

- No (Нет) — исходный код не выводится, выводятся только дизассемблрованные инструкции.
- Yes (Да) — перед первой дизассемблированной инструкцией, со ответствующей данной строке, выводится строка исходного кода. Область устанавливается в данный режим, если исходный модуль написан на языке высокого уровня.
- **Both** (Оба) — для тех строк, которым соответствует исходный код, дизассемблированные строки заменяются строками исходного текста. В противном случае выводятся дизассемблированные инструкции. Используйте этот режим, когда вы отлаживаете модуль на

ассемблере и хотите видеть строку исходного текста, а не соответствующую дизассемблированную инструкцию. Область устанавливается в данный режим вывода, если текущим модулем является исходный модуль ассемблера.

Thread

Позволяет выбрать нить, выполнение которой вы хотите отладить. Открывает диалоговое окно **Pick a Thread**, из которого вы можете выбрать конкретную нить программы.

OS Exceptions

Позволяет выбрать исключительные ситуации операционной системы, которые вы хотите обрабатывать.

New EIP

Изменяет текущий адрес программы, подсвеченный в области кода (в TDW команда называется **New CS:IP**). При возобновлении выполнения программы оно начинается по этому адресу. Эта команда полезна, когда нужно пропустить некоторые машинные инструкции, но использовать ее нужно аккуратно, так как она может вызвать нестабильность системы.

Assemble

Ассемблирует инструкцию, заменяя инструкцию по текущему адресу. Используется для внесения в программу минимальных изменений. Команда выводит диалоговое окно **Enter Instruction to Assemble**, где вы можете ввести выражение для ассемблирования. Если вы начнете набор в области кода, данная команда вызывается автоматически.

I/O

Эта команда TDW считывает или записывает значения в пространство адресов ввода-вывода ЦП и позволяет вам проверить содержимое регистров ввода-вывода и записать в них значения. При этом выводится меню, показанное ниже:

```
In byte         Ввести байт из порта
Out byte        Вывести байт в порт
Read byte       Прочитать байт из порта
Write byte      Записать байт в порт
```

Учтите, что эти команды могут нарушить нормальную работу устройств.

Область регистров и флагов

В области регистров (верхняя область справа от области кода) выводится содержимое регистров процессора. Вид этой области зависит от отладчика (TD32 или TDW). По умолчанию TDW выводит 13 16-разрядных регистров, а TD32 — всегда выводит 15 регистров процессора 80386 и старше.

С помощью команд **SpeedMenu** области регистров вы можете модифицировать или сбрасывать содержимое регистров. Команда **Increment** добавляет 1 к текущему подсвеченному регистру, **Decrement** вычитает 1 из содержимого текущего подсвеченного регистр, а **Change** позволяет изменить содержимое регистра, выводя диалоговое окно **Enter New Value** для ввода нового значения. Последняя команда вызывается автоматически, если вы начинаете набор в области регистров.

Команда **Registers 32-bit**, доступная только в TDW, переключает вывод регистров с 16-битовых на 32-битовые (сегментные регистры остаются 16-битовыми).

Область флагов

В области флагов показано значение каждого флага ЦП. Список различных флагов и то, как они выводятся в области флагов, показан в следующей таблице:

Буква в области	Название флага
c	Флаг переноса
z	Флаг нуля
s	Флаг знака
o	Флаг переполнения
p	Флаг четности
a	Флаг дополнительного переноса
i	Флаг разрешения прерывания
d	Флаг направления

SpeedMenu этой области содержит содержит команду **Toggle**, переключающую значение подсвеченного флага между 0 и 1.

Область дампа

В этой области выводится в шестнадцатеричном виде содержимое области памяти. В левой части каждой строки

показан адрес (в виде «сегмент:смещение» или 32-разрядного адреса). Порядок регистров в области **Dump** имеет вид: **DS**, **ES**, **SS**, **CS**. Справа от адреса выводятся значения элементов данных в выбранном формате.

SpeedMenu области **Dump** содержит команды для перемещения по области, модификации содержимого, перемещению по указателям, задания формата вывода и работы с блоками памяти.

Goto

Выводит диалоговое окно **Enter Address to Position To**, где вы можете ввести выражение, при вычислении которого получается адрес памяти, доступный программе.

Search

Ищет строку символов или список байт, начиная с адреса, указанного курсором.

Next

Ищет следующий экземпляр элемента, заданного в команде поиска.

Change

Позволяет модифицировать байты по текущему месту расположения курсора. При выводе в формате ASCII или шестнадцатеричном виде запрашивается список байт, в противном случае — элемент текущего формата вывода.

Follow

Открывает меню с командами, позволяющими проверить данные по адресам указателей **near** и **far**. TD32 содержит команды для 32-разрядной адресации.

Команда **Near Code** этого меню интерпретирует слово под курсором в области данных, как смещение в текущем сегменте кода (как это задается регистром **CS**). Область кода становится текущей областью и позиционируется на данный адрес.

Команда **Far Code** интерпретирует двойное слово под курсором в области данных, как адрес дальнего типа (сегмент и смещение). Область кода становится текущей и позиционируется на данный адрес.

Команда **Offset to Data** позволяет вам следовать по цепочке указателей размером в слово (ближнего типа, где используется

только смещение). Область данных устанавливается в соответствии со смещением, заданным словом в памяти по текущей позиции курсора.

Команда **Segment:Offset to Data** позволяет следовать по цепочке указателей дальнего типа размером в двойное слово (где используется сегмент и смещение). Область данных устанавливается в соответствии со смещением, заданным двойным словом в памяти по текущей позиции курсора.

Команда **Base Segment:0 to Data** интерпретирует слово под курсором, как адрес сегмента, и позиционирует область данных на начало сегмента.

Previous

Восстанавливает адрес области данных в адрес, который был до последней команды, явно изменившей значение текущего адреса. Использование клавиш стрелок и клавиш перемещения курсора не приводит к запоминанию позиции. Отладчик поддерживает стек из пяти последних адресов, поэтому вы можете вернуться назад после многократного (< 5) использования команд локального меню **Follow** или команды **Goto**.

Display As

Позволяет выбирать формат вывода в области данных. Вы можете выбирать один из форматов данных, использующихся в языке Си, Pascal или ассемблер. Эти форматы можно выбрать из меню. Команда **Byte** устанавливает область данных в режим вывода шестнадцатеричных байтовых данных. **Word** устанавливает область данных в режим вывода шестнадцатеричных слов. **Long** задает режим вывода длинных шестнадцатеричных целых чисел. **Comp** устанавливает режим вывода 8-байтовых целых чисел. Выводится десятичное значение числа. **Float** устанавливает режим вывода 6-байтовых чисел с плавающей точкой. Выводится значение числа с плавающей точкой в научном представлении. **Double** выводит 8-байтовые числа с плавающей точкой. Выводится значение числа в научном представлении. **Extended** устанавливает режим вывода 10-байтовых чисел с плавающей точкой в научном представлении.

Block

Позволяет работать с блоками памяти. Вы можете перемещать, очищать, присваивать значения блокам памяти, а также записывать и считывать блоки памяти из файлов на диске. По данной команде на экран выводится всплывающее меню. Команда **Clear** этого меню устанавливает непрерывный блок в памяти в значение 0. Адрес блока и число байт, которые требуется очистить, запрашиваются в выводимой подсказке. **Move** копирует блок памяти из одного адреса в другой. Адреса исходного и целевого блока, а также число копируемых байт, будут запрашиваться в подсказке. **Set** присваивает непрерывному блоку в памяти конкретное байтовое значение. Адрес блока, число байт, которым требуется присвоить значение, а также само значение запрашиваются в подсказке. **Read** считывает все содержимое или часть файла в блок памяти. Вам выводится подсказка для ввода имени считываемого файла, затем адреса, куда требуется считать информацию, и числа считываемых байт. **Write** записывает блок памяти в файл. Выводится подсказка для ввода имени файла, куда требуется записать данные, затем блока памяти, который нужно записать, и число считываемых байт.

Область стека

Эта область показывает шестнадцатеричное содержимое программного стека. Текущий указатель стека отмечается указателем >. **SpeedMenu** этой области содержит команды **Goto**, **Origin**, **Follow**, **Previous** и **Change**, аналогичные описанным выше командам.

Область селектора

В этой области (только для TDW) выводится список селекторов защищенного режима и указывается некоторая информация для каждого из них. Селектор может быть допустимым или нет. Допустимый селектор указывает на ячейку таблицы дескрипторов защищенного режима, соответствующего адресу памяти. Если селектор недопустим, то он не используется. Для допустимого селектора в области выводится следующее:

- являются ли содержимым данные или код;
- загружена ли область памяти, на которую ссылается селектор (присутствует в памяти) или разгружена (выведена на диск);

- длина сегмента памяти, на которую ссылается селектор (в байтах).

Если селектор ссылается на сегмент данных, то имеется дополнительная информация по полномочиям доступа (**Read/Write** — Чтение/ Запись или **Read only** — только чтение) и направление расширения сегмента в памяти (**Up** — вверх или **Down** — вниз).

Локальное меню области можно использовать для перехода к новому селектору или просмотра содержимого подсвеченного. В зависимости от характера данных, содержимое выводится в области кода или области дампа.

Команда **Selector** выводит подсказку для ввода селектора, который нужно вывести в области. Для ввода селектора вы можете использовать полный синтаксис выражений. Если вы вводите числовое значение, то TDW подразумевает, что оно десятичное (если вы не используете синтаксис текущего языка для указания того, что значение является шестнадцатеричным).

Другим методом ввода значения селектора является вывод окна **CPU** и проверка содержимого сегментных регистров. Если регистр содержит интересующий вас селектор, то вы можете ввести имя регистра с предшествующим символом подчеркивания (_). Например, вы можете задать имя сегментного регистра данных, как **_DS**.

Команда **Examine** выводит содержимое области памяти, на которую ссылается текущий селектор, и переключается в область, где выводится содержимое. Если селектор указывает на сегмент кода, то содержимое выводится в области кода. Если содержимое представляет собой данные, то оно выводится в области данных.

Окно Dump

В этом окне выводится в непосредственном виде дамп любой области памяти. Оно работает так же, как область данных окна **CPU**.

```
[*]  Dump                                              3
ds:0000 CD 20 00 A0 00 9A F0 FE = & U**
ds:0008 1B 02 B2 01 22 31 7C 01 <.^% .`
ds:0010 22 31 88 02 52 2B E2 1D * X 4-#
ds:0018 01 01 01 00 03 FF FF FF
```

С помощью команды **View Another Dump** вы можете одновременно открыть несколько окон **Dump**.

Окно Registers

В окне **Registers** выводится содержимое регистров и флагов центрального процессора. Оно работает, как сочетание областей регистров и флагов в окне **CPU** и имеет те же команды.

Отладка в Windows

Дополнительная сложность программ для Windows вызывает появление новых категорий ошибок. Turbo Debugger имеет ряд средств, которые помогут вам найти ошибке в программе для Windows.

Регистрация сообщений

Окно **Windows Messages** имеет ряд команд для трассировки и проверки получаемых программой оконных сообщений. С его помощью вы можете устанавливать точки останова по сообщениям (выполнение программы будет приостанавливаться при получении сообщения конкретным окном). Вы можете также регистрировать получаемые окном сообщения. Данное окно открывается командой **View Message** и имеет три области:

- область выбора окна
- область класса сообщения
- область регистрации.

Задание окна

Чтобы регистрировать сообщения для конкретного окна, задайте это окно, отслеживаемые сообщения и действия, выполняемые отладчиком при их получении: прерывание выполнения (**Break**) или регистрация (**Log**).

Чтобы задать окно в TD32, используйте имя оконной процедуры, которая обрабатывает сообщения окна. Для этого с помощью команды **Add** в **SpeedMenu** области выбора окна откройте диалоговое окно **Add Window Procedure to Watch** (или наберите непосредственно ее имя в области). Затем наберите имя процедуры в поле ввода **Window Identifier** и нажмите **Enter**. Эту процедуру вы можете повторить для каждого окна, сообщения которому вы хотите отслеживать.

В TDW окно можно задать с помощью описателя окна или оконной процедуры, обрабатывающей его сообщения. В любом случае следует использовать диалоговое окно **Add Window** или **Handle to Watch**. Для его вывода выберите команду **Add** в **SpeedMenu** области выбора окна или наберите имя непосредственно в этой области. Кнопки **Identify By** этих окон позволяет вам выбрать способ спецификации окна. Это меню позволяет также отменить выбор окна. Для этого используются команды **Remove** (**Ctrl+R**) и **Delete All** (**Ctrl+D**).

Задание отслеживаемых сообщений

После задания окна Turbo Debugger по умолчанию перечисляет в области регистрации сообщения все сообщения **WM_**. Чтобы сократить число отслеживаемых сообщений, используйте диалоговое окно **Set Message Filter**, которое выводится командой **Add** в **SpeedMenu** области класса сообщения. Это окно позволяет задать класс сообщений или индивидуальные имена сообщений.

Чтобы задать конкретное сообщение для окна в области выбора окна, откройте диалоговое окно **Set Message Filter** и с помощью кнопки с зависимой фиксации выберите один из следующих классов сообщений:

All Messages

Все оконные сообщения.

Mouse

Сообщения, генерируемые событием «мыши».

Window

Сообщения, генерируемые администратором окон.

Input

Сообщения, генерируемые клавиатурным событием, или обращением пользователя к меню **System**, полосе прокрутки или блоку изменения размера.

System

Сообщения, генерируемые изменениями в масштабе системы.

Initialization

Сообщения, генерируемые при создании в приложении диалогового окна.

Clipboard

Сообщения, генерируемые при обращении пользователя к буферу Clipboard.

DDE

Сообщения динамического обмена данными, генерируемые при обмене данными между приложениями Windows.

Non-client

Сообщения, генерируемые Windows для обслуживания неклиентной области окна приложения.

Other

Любые сообщения, не попадающие в предыдущие категории (например, сообщения MDI).

Single Message

Позволяет вам задать конкретное отслеживаемое сообщение.

Чтобы регистрировать одно сообщение, выберите **Single Message** и введите в поле ввода **Single Message Name** имя сообщения или его номер. Если вы хотите регистрировать для конкретного окна несколько классов или сообщений, то:

- задайте конкретный класс или имя сообщения;
- выберите в **SpeedMenu** области классов сообщений команду **Add**;
- в определение отслеживаемых сообщений добавьте классы или имена сообщений.

Задание действия по сообщению

После спецификации окна и отслеживаемых сообщений нужно задать действие, выполняемое при поступлении сообщения. Turbo Debugger предусматривает в диалоговом окне **Set Message Filter** две кнопки **Action**: **Break** (приостановка выполнения программы) и **Log** (регистрация сообщения вы области регистрации окна **Windows Messages**). **Break** фактически означает установку точки останова по сообщения.

Если вы регистрируете сообщения для нескольких окон, не регистрируйте все сообщения. Большое число передаваемых между Windows и Turbo Debugger сообщений может привести к краху системы.

Отладка библиотек DLL

Динамически компонуемая библиотека **DLL** — это библиотека подпрограмм и ресурсов, компонуемая с приложением Windows на этапе выполнения. Это позволяет подпрограммам использовать одну копию подпрограмм, ресурсов и драйверов устройств. Когда приложению требуется доступ к **DLL**, Windows проверяет, загружена ли **DLL** в память. Если это так, то вторая копия не загружается.

DLL может загружаться программой в память двумя различными способами:

- при загрузке программы (**DLL** загружается при статической компоновке ее с программой с помощью утилиты **IMPLIB**);
- когда ваша программа обращается с вызовом **LoadLibrary**.

Выполнение DLL по шагам

При пошаговом выполнении функции **DLL** Turbo Debugger загружает идентификатор **DLL**, исходный код **DLL** в окно Windows и позиционирует курсор на вызываемую подпрограмму. Однако, перед загрузкой исходного кода в окно **Module** должны удовлетворяться следующие условия:

- **DLL** должна компилироваться с отладочной информацией.
- Файл .**DLL** должен находиться в том же каталоге, что и файл **.EXE** программы.
- Должен быть доступен исходный код **DLL**.

Turbo Debugger ищет исходный код **DLL** также, как и исходный код программ. Если **DLL** не содержит отладочной информации, то отладчик не может найти исходный код **DLL** и открывает окно **CPU**.

При отладке функции **DLL** и прохождении с помощью **F7** или **F8** оператора **return** ваша программа может начать работать, хотя вы нажали **F9**. Такое поведение типично при отладке **DLL**,

вызванной из программы без отладочной информации, или когда **DLL** возвращает управление через функциональный вызов Windows.

Если вы отлаживаете код запуска **DLL**, перед загрузкой **DLL** установите точку останова на первой строке программы. Это обеспечит приостановку программы при возврате и **DLL**.

Доступ к DLL и исходному коду модулей

Хотя Turbo Debugger обеспечивает прозрачное пошаговое выполнение функций **DLL**, вам может потребоваться доступ к **DLL** до того, как программа ее вызовет (например, в ней нужно установить точки останова или задать отслеживаемые выражения). Для доступа к выполняемому модулю, отличному от текущего загруженного, откройте с помощью команды **View Modules** (**F3**) диалоговое окно **Load Module Source or DLL**. Это диалоговое окно перечисляет все исходные модули, содержащиеся в текущем загруженном выполняемом файле. Блок списка **DLL & Programs** показывает все файлы **.DLL** и **.EXE**, загруженные Windows. (При работе с TDW в нем также выводятся все загруженные файлы **.DRV** и **.FON**.)

Символом точки (**.**) отмечены **DLL**, которые могут загружаться в Turbo Debugger (а также **DLL** с отладочной информацией и исходным кодом). Звездочка (*) показывает, что модуль загружен отладчиком. Так как ваши программы могут загружать **DLL** с помощью вызова **LoadLibrary**, в блоке списка могут показываться не все **DLL**.

Если вам нужен другой модуль исходного кода, подсветите нужный модуль в списке **Source Module** и используйте кнопку **Load** (или дважды щелкните на имени модуля «мышью»). Turbo Debugger открывает окно **Module** и выводит исходный код данного модуля.

Для доступа к выполняемому файлу, отличному от текущего, откройте диалоговое окно **Load Module Source or DLL Symbols** (**F3**), подсветите в блоке списка нужный файл и выберите командную кнопку **Symbol Load**. Turbo Debugger открывает окно **Module** с исходным кодом первого модуля выполняемого файла.

Чтобы добавить **DLL** к списку, откройте указанное диалоговое окно, активизируйте поле ввода **DLL Name** и введите

имя соответствующей **DLL**. Чтобы добавить **DLL** к списку, нажмите кнопку **Add DLL**.

При выполнении по шагам функции **DLL** отладчик автоматически загружает таблицу идентификаторов и исходный код этой **DLL**. Чтобы предотвратить это, откройте диалоговое окно **Load Module Source or DLL Symbols (F3)**, подсветите в списке нужную **DLL**, выберите кнопку **No** и щелкните «мышью» на **OK**. Turbo Debugger будет выполнять вызовы **DLL** как одну команду.

Отладка кода запуска DLL

Когда ваша программа загружает **DLL**, выполняется код запуска **DLL**. По умолчанию Turbo Debugger не выполняет по шагам этот код. Однако, если вам нужно проверить корректность загрузки **DLL**, то нужно отладить код запуска. Отладчик позволяет отлаживать два вида такого кода: код инициализации, непосредственно следующий за **LibMain** (по умолчанию) и скомпонованный с **DLL** код ассемблера. Этот код инициализирует процедуры запуска и эмулирует математические пакеты (этот режим отладки выбирается параметром **-l** командной строки отладчика).

Чтобы начать отладку кода запуска **DLL**, нужно перезагрузить программу (**Run Program Reset** или **F2**), а затем выполнить следующие шаги:

- вывести диалоговое окно **Load Module Source or DLL Symbols (F3)**;
- подсветите в блоке списка **DLL & Programs DLL**, код запуска которой вы хотите отладить;
- выберите кнопку с зависимой фиксацией **Debug Startup** (если нужной **DLL** в списке нет, добавьте ее как описано выше);
- повторите эти шаги, если нужно задать отладку для нескольких **DLL**;
- для перезагрузки приложения выберите команду **Run Program Reset** или **F2**.

При отладке имейте в виде следующее:

- Перед перезагрузкой текущего приложения выполняйте до конца код запуска **DLL**, иначе Windows может зависнуть.

- Установка точек останова на первой строке приложения или первой строке после вызова **LoadLibrary** гарантирует возврат управления в Turbo Debugger.
- После завершения отладки кода запуска нажмите **F9**, чтобы пройти его до конца и вернуться в приложение.

Отладка мультинитевых программ

Окно **Thread**, которое открывается по команде **View Thread**, поддерживает мультинитевую среду Windows NT. Это окно содержит три области: списка нитей, детализации и информационную.

В информационной области перечисляется общая информация о нити. Поле **Last** указывает последнюю нить, выполненную перед передачей управления в Turbo Debugger; поле **Current** показывает нить, которая выводится в окнах отладчика; поле **Total** — общее число активных программных нитей, а поле **Notify** — **Yes** или **No** для статуса **Notifu** или **Termination** отдельных нитей. Общий статус устанавливается с помощью команды **All Threads**.

Область нитей

В этой области перечисляются все активные нити программы, идентифицируемые по номеру нити (назначаемому Windows NT) и имени. Turbo Debugger генерирует имя нити, когда ваша программа создает нить. Первая создаваемая нить называется **Thread 1**, затем **Thread 2** и т.д. Это имя можно изменить.

Окно **Thread** содержит единое **SpeedMenu**, которое активизируется из всех областей и содержит перечисленные ниже команды.

Options

Открывает диалоговое окно **Thread Options**, позволяющее задать параметры отдельных нитей. Кнопка **Freeze** этого окна позволяет «замораживать» и «размораживать» индивидуальные нити. Включение этой кнопки означает, что нить выполняться не будет. Для выполнения программы необходима хотя бы одна активная нить. Кнопка **Notify or Tremination** позволяет задать, должен ли отладчик уведомлять вас о завершении текущей (подсвеченной) нити (он генерирует сообщение и активизирует

окно **Module** и **CPU** с текущим адресом программы). Чтобы задать уведомление для всех нитей, используйте команду меню **All Threads**. Поле ввода **Thread Name** позволяет изменить имя текущей нити.

Make Current

Команда **Make Current** позволяет сменить нить, обрабатываемую отладчиком. Подсветите в области **Threads List** нить, которую вы хотите проверить, и нажмите **Ctrl+M** (или выберите **Make Current**).

Inspect

Открывает окно **Module** или **CPU**, которое показывает для подсвеченной нити текущую точку выполнения. Этой команде эквивалентна клавиша **Enter**.

All Threads

Открывает меню, команды которого относятся ко всем нитям программы. Это команды **Thaw**, **Freeze**, **Enable Exit Notification** и **Disable Exit Notification**.

Step

Позволяет переключаться между **All** и **Single**. При выборе **All** клавиши **F7** и **F8** приводят к выполнению всех нитей программы, а **Single** позволяет выполнять только одну нить.

Область детализации

В этой области выводится подробная информация о нити, подсвеченной в области списка нитей. Первая строка показывает статус подсвеченной нити (приостановлена или выполняется) и ее приоритет. Операционная система устанавливает 5 различных приоритетов (от -2 до 2). Вторая строка показывает текущую точку выполнения нити, а третья (если она есть) — как получил управление отладчик.

Трассировка исключительных ситуаций операционной системы

В TD32 команда **OS Exceptoions** (в **SpeedMenu** области кода окна **CPU**) открывает диалоговое окно **Specify Exception Handling**, в котором вы можете задать, как Turbo Debugger должен

обрабатывать исключительные ситуации операционной системы, генерируемые программой.

В блоке списка **Exceptions** показаны все исключительные ситуации операционной системы, обрабатываемые отладчиком. Для каждой из них вы можете задать обработку отладчиком или программой обработки исключительных ситуаций. По умолчанию они обрабатываются в Turbo Debugger. Он приостанавливает программу и активизирует окно **Module** или **CPU**, устанавливая курсор на соответствующую строку кода.

Чтобы изменить это заданное по умолчанию поведение, откройте окно **Specify Exception Handling**, подсветите исключительную ситуацию, которую вы хотите обрабатывать вы программе, и щелкните «мышью» на кнопке с независимой фиксацией **User Program**.

Если вы хотите, чтобы программа обрабатывала все исключительные ситуации операционной системы, используйте кнопку **User All**.

Задание пользовательских исключительных ситуаций

Поля ввода **Range Low** и **Range High** окна **Specify Exception Handling** позволяет задать исключительные ситуации операционной системы, определенные пользователем. По умолчанию оба эти поля устанавливаются отладчиком в 0. Введите в поле **Range Low** шестнадцатеричное значение, генерируемое исключительной ситуацией. Если определяется несколько исключительных ситуаций, в поле **Range High** введите также максимальный номер определенной пользователем исключительной ситуации.

Память и списки модулей

В TDW вы можете записать в окно **Log** содержимое глобальной и локальной динамической памяти или список используемых программой модулей. Окно **Windows Information** (доступное с помощью команды **Display Windows Info** в **SpeedMenu** окна **Log**) позволяет выбрать тип выводимого списка и где вы хотите его начать.

Глобальная динамически распределяемая область памяти — это память, которую Windows делает доступной для всех приложений. Эта память используется при распределении

ресурсов. Чтобы увидеть список объектов данных в глобальной области, выберите в **Windows Information** кнопку с зависимой фиксацией **Global Heap** и щелкните «мышью» на **OK**. Объекты данных выводятся в окне **Log**.

Кнопка с зависимой фиксацией **Start At** позволяет вам выводить список с нужного места динамически распределяемой области (с начала, с конца или с места, заданного начальным описателем, устанавливаемым вызовом **GlobalAlloc**).

Чтобы вывести список всех задач и модулей **DLL**, загруженных в Windows, выберите в диалоговом окне **Windows Information** кнопку **Module List**, затем **OK**. Модули будут перечисляться в окне **Log**.

Отладка объектно-ориентированных программ

В Turbo Debugger предусмотрен ряд средств для отладки объектно-ориентированных программ C++.

Окно Hierarchy

Окно **Hierarchy** (открываемое командой **View Hierarchy**) служит для проверки иерархии объектов или классов, которая выводится в графическом виде.

```
Область классов               Область иерархии

 [*]    Class Hierarchy                    3     [^][v]
Devi e                 Point
GlowGauge                          Rectangle
HorzArrow                                   Device
HorzBar                                     TextWindow
LinearGauge  Range
Point                Device
Range                         GlowGauge
Rectangle
Screen       Parents of Device
TextWindow               Range
VertArrow                Rectangle
VertBar                             Point
                                            Screen
             <                                        >
```

Область порождающих классов

Это окно выводит наследование классов C++ и, в зависимости от использования в программе множественного наследования, состоит из трех областей.

Область классов

Эта область выводит в алфавитном порядке список всех классов, используемых в загруженном модуле. Справа представлена детальная информация по подсвеченному здесь классу. Для быстрого поиска класса используется средство инкрементального поиска. Если вы начнете набирать здесь имя класса, отладчик подсвечивает имя, начинающееся с набранных символов.

SpeedMenu этой области содержит две команды. Команда **Inspect** (или клавиша **Enter**) открывает для текущего класса окно **Class Inspector**. Команда **Tree** активизирует область иерархии, подсвечивая текущий класс.

Область иерархии

Здесь выводятся классы загруженного модуля и их иерархии. Базовые классы размещаются по левому полю области. Классы, наследующие из нескольких базовых классов, отмечаются звездочками (**), а все другие классы, являющиеся частью той же группы множественного наследования — одной.

Локальное меню этой области содержит две команды. Команда **Inspect** (или клавиша **Enter**) открывает для подсвеченного класса окно **Class Inspector**. При отладке программ C++ с множественным наследованием здесь доступна также команда **Parents**, включающая и выключающая вывод области порождающих классов окна **Hierarchy**.

Область порождающих классов

Эта область выводится только для программ с множественным наследованием и при ее разрешении. Для классов, полученных путем множественного наследования, она выводит все производные классы. **SpeedMenu** этой области содержит единственную команду **Inspect**. При ее выборе (или нажатии **Enter**) для подсвеченного класса выводится окно **Class Inspector**.

Окна Class Inspector

Эти окна позволяют вам вывести детальную информацию по классам C++. Чтобы открыть это окно, выведите окно **Hierarchy**, подсветите класс и нажмите **Enter**.

```
[Ч]  Class LinearGauge       4
int Range       ::Low
int Range       ::High
int Screen      ::MaxX
class Range *Range::ctr()
int Range::GetValue()
int Range::GetLow()
int Range::GetHigh()
```

Данное окно содержит две области. В верхней области выводится информация о элементах данных и их типах, в нижней — о функциях-элементах и возвращаемых типах. Однако это окно не отражает данных конкретного экземпляра. Если вы хотите проверить аргументы функцию-элемента, подсветите ее и нажмите **Enter**. Откроется окно **Function Inspector**.

Если подсвеченный элемент данных представляет собой указатель на класс, то нажатие **Enter** открывает другое окно **Class Inspector**. Таким образом вы можете проверять сложные вложенные классы. Как и в случае других окон **Inspector** клавиша **Esc** закрывает текущее окно **Inspector**, а **Alt+F3** закрывает их все.

SpeedMenu каждой области данного окна содержит три команды, которые в каждой области ведут себя несколько по разному.

Inspect

В области элементов данных открывает для подсвеченного элемента данных окно **Inspector**. В области функций-элементов команда открывает для подсвеченной функции окно **Function Inspector**. Для вывода исходного кода функции позиционируйте курсор на адрес функции-элемента и нажмите **Enter**. Откроется окно **Module**.

Hierarchy

Во всех областях открывает окно **Hierarchy** для текущего подсвеченного класса.

Show Inherited

В каждой области переключается между **Yes** (по умолчанию) и **No**. При установке в **Yes** Turbo Debugger показывает для подсвеченного класса все функции-элементы или элементы данных, включая наследуемые. В противном случае выводятся только элементы данного класса.

Окно Object Inspector

Это окно используется для просмотра структуры и значений конкретного экземпляра класса. Чтобы открыть данное окно, поместите курсор на конкретный экземпляр класса (в окне **Module**) и нажмите **Ctrl+I**.

```
[*]  Inspecting tw                              3
@75C6:01E8
Screen::MaxX          500                     (0x1F4)
Screen::MaxY          512                     (0x200) v
Screen::Convert       @0000:0000
Screen::VertVtoA      @0000:0000
Screen::VertAtoV      @0000:0000
class TextWindow
```

Данное окно содержит три области. Область элементов данных (верхняя) показывает текущие значения элементов данных объектов. Окно функций-элементов (среднее) выводит текущие значения и адреса функций-элементов объекта. Область типов показывает тип подсвеченного элемента данных или функции-элемента.

SpeedMenu верхних двух областей содержат идентичные команды, только область элементов данных содержит дополнительную команду **Change**.

Range

Позволяет вам задать диапазон выводимых элементов массива. Если подсвеченный элемент не является массивом или указателем, то команда недоступна.

Change

Позволяет изменить значение подсвеченного элемента данных.

Methods

Переключается между **Yes** (по умолчанию) и **No**. В состоянии **Yes** отладчик выводит среднюю область окна **Object**

Inspector с перечислением функций-элементов. **No** отменяет вывод средней области.

Show Inherited

Также переключается между **Yes** и **No**. В состоянии **Yes** показываются все функции-элементы, определенные в классе и наследуемые. **No** позволяет вывести только функции-элементы, определенные в классе.

Inspect

Открывает для текущего подсвеченного элемента окно **Inspector**. Проверка функции-элемента открывает окно **Module**, где курсор позиционируется на определяющий эту функцию код.

Descend

Работает аналогично команде **Inspect SpeedMenu**, но заменяет текущее окно **Inspector**. Это уменьшает число открытых на экране окон **inspector**.

New Expression

Используется для проверки различных выражений. Данные в текущем окне **Inspector** заменяются данными нового вводимого выражения.

Type Cast

Позволяет задавать для текущего подсвеченного элемента данные различного типа. Эта команда полезна, если идентификатор не имеет информации о типе и для явного задания типа указателей.

Hierarchy

Открывает окно **Hierarchy** с наследованием текущего класса.

Отладка резидентных программ и драйверов устройств

С помощью TD.EXE вы можете отлаживать не только обычные выполняемые файлы, но также резидентные в памяти программы (TSR) и драйверы устройств. Вы можете кроме того выполнять сам отладчик, как резидентную программу (в то время, как работаете на уровне DOS или запускаете другие программы).

Что такое резидентная программа?

Резидентными (TSR) называют такие программы, которые остаются в оперативной памяти после того, как они завершат управление. В Borland Си и C++, предусмотрена специальная функция **geninterrupt**, которая выдает такое программное прерывание.

Резидентная программа состоит из двух частей — рабочей части и резидентной части. Рабочая часть выполняет загрузку резидентной части в память и устанавливает вектор прерываний, который определяет характер вызова резидентной в памяти программы. Если резидентная программа должна вызываться с помощью программного прерывания, то рабочая часть программы помещает адрес резидентной части кода в соответствующий вектор прерывания. Если резидентная программа должна вызываться с помощью оперативной клавиши, то резидентная часть должна модифицировать обработчик прерывания DOS для обработки нажатия соответствующих клавиш (клавиши) на клавиатуре.

Когда рабочая часть завершает выполнение, она вызывает функцию DOS, которая позволяет части файла **.EXE** оставаться резидентной в оперативной памяти после завершения выполнения программы. Рабочая часть резидентной программы знает размер резидентной части, а также ее адрес в памяти, и передает эту информацию DOS. Операционная системе DOS при этом резервирует специальный блок памяти, но может свободно записывать информацию в незащищенную часть памяти. Таким образом, резидентная часть остается в памяти, а рабочая часть может быть «затерта».

Тонкость отладки резидентных программ состоит в том, что вы должны иметь возможность отлаживать и резидентную, и рабочую часть программы. Когда выполняется файл **.EXE**, то выполняется только код рабочей части TSR. Поэтому, когда вы как обычно запускаете отладчик, задав имя файла, вы видите выполнение только рабочей части кода программы: то, как он устанавливает резидентную часть и обработчики прерываний. Чтобы отлаживать резидентную часть, вы должны задать точку останова и сделать резидентным сам отладчик.

Отладка резидентной в памяти программы

Отладка рабочей части резидентной программы эквивалентна отладке любого другого файла. Новое появляется только тогда, когда вы начинаете отлаживать резидентную часть. Давайте рассмотрим процесс отладки резидентной программы.

С помощью Turbo Debugger вы можете отлаживать драйвер клавиатуры. При этом для перемещения по отладчику пользуйтесь «мышью».

- При компиляции или ассемблировании резидентной программы обеспечьте наличие в ней отладочной информации.
- Запустите отладчик и загрузите программу.
- Установите точку останова в начале резидентной части кода.
- С помощью команды **Run Run** запустите рабочую часть программы.
- Отладьте рабочую часть программы с помощью обычных методов.
- Затем выйдите из TSR. Резидентная часть остается в памяти.
- Чтобы сделать резидентным отладчик, выберите команду **File Resident**. На TSR это не повлияет. После этого вы можете вернуться в DOS и вызвать TSR.
- В командной строке DOS нажмите оперативные клавиши вызова резидентной программы и работайте с ней как обычно.
- Выйдите из TSR. Теперь выполняется резидентная часть TSR, и отладчик обнаруживает точку останова. Вы можете отлаживать резидентный код.

Второй метод отладки резидентной части TSR предусматривает выполнение ее из командной строки DOS и использование окна **CPU** отладчика для отладки содержащей TSR области ОЗУ.

- Скомпилируйте программу с отладочной информацией.

- Используйте утилиту **TDSTRIP** для удаления из программы таблицы идентификаторов и помещения ее в файл **.TDS**.
- Запустите TSR из командной строки.
- Запустите утилиту **TDMEM**, которая выводит схему использования памяти. Запомните адрес сегмента, где загружена резидентная часть вашей программы.
- Загрузите отладчик и с помощью команды **File Symbol Load** загрузите таблицу идентификаторов TSR (файл **.TDS**).
- Установите в начале резидентной части TSR точку останова.
- Чтобы сделать отладчик резидентным, выберите команду **File Resident**.
- В командной строке DOS выполните резидентную часть TSR, нажав ее оперативную клавишу, и работайте с программой как обычно. При обнаружении точки останова отладчик приостанавливает TSR в начале резидентной части. Чтобы облегчить работу, синхронизируйте таблицу идентификаторов с кодом в памяти. Идентификаторы в таблице отстоят друг от друга на корректное число байт, но абсолютный адрес первого идентификатора не определен, так как DOS загрузила резидентную программу по адресу в памяти, отличном от того, с которым она ассемблировалась. Поэтому, чтобы найти первый идентификатор в памяти, используйте команду **File Table**.
- Используйте команду **File Table Relocate** для помещения первого идентификатора из таблицы идентификаторов в соответствующую ячейку памяти. Таким образом, имеющаяся информация об идентификаторах будет соответствовать вашему коду (программе). Для этого в ответ на подсказку отладчика задайте адрес сегмента Seg вашей резидентной программы, который определен с помощью утилиты TDMEM, плюс шестнадцатеричное значение 10 (для PSP размером 256 байт). Дизассемблированные из памяти операторы синхронизированы с информацией из таблицы

идентификаторов. В случает наличия исходного файла исходные операторы выводятся на той же строке, что и информация из таблицы идентификаторов.

- Для перехода к сегменту оперативной памяти, где находится ваша резидентная программа, используйте команду **Goto** (клавиши **Ctrl-G**). Это можно сделать, используя адрес сегмента вашей программы TSR, за которым следует смещение 0000H, или с помощью перехода на конкретную метку вашей программы.
- Отладьте резидентную часть программы.

Что такое драйвер устройства?

Драйвер устройства — это набор подпрограмм, используемых операционной системой DOS для управления на нижнем уровне функциями ввода-вывода. Устанавливаемые драйверы устройств (в отличие от драйверов, встроенных в DOS) устанавливаются с помощью включения соответствующих строк, например:

```
device = clock.sys
```

в файл **CONFIG.SYS**. Когда DOS выполняет операцию ввода-вывода для отдельного символа, она просматривает связанный список заголовков устройств, выполняя поиск устройства с соответствующим логическим именем (например, **COM1**). В случае драйверов блочно-ориентированных устройств, таких, как драйвер диска, DOS отслеживает, сколько установлено драйверов блочно-ориентированных устройств, и обозначает каждый из них буквой: **A** — первый установленный драйвер устройства, **B** — второй и т.д. Когда вы, например, ссылаетесь на дисковод **C**, DOS знает, что нужно вызвать драйвер третьего блочно-ориентированного устройства.

Связанный список двух заголовков драйвера содержит смещение двух компонентов самого драйвера устройства: подпрограмму функции и подпрограмму обработки прерывания.

Когда DOS определяет, что требуется вызвать данный драйвер устройства, она вызывает драйвер дважды. При первом вызове драйвера DOS общается с подпрограммой функции и передает ей указатель на буфер в памяти, который называется заголовком запроса. Этот заголовок запроса содержит информацию о том, какие функции требует выполнить DOS от

драйвера устройства. Подпрограмма функции просто сохраняет данный указатель для последующего использования. При втором вызове драйвера устройства DOS вызывает подпрограмму обработки прерывания, которая выполняет реальные функции, заданные DOS в заголовке запроса, например, пересылку символов с диска.

В заголовке запроса с помощью байта, который называется кодом команды, определяется, что должен делать драйвер устройства. Код команды определяет одну из предопределенных операций из набора операций, которые должны выполнять все драйверы устройств. Набор кодов команд (операций) для драйверов символьно-ориентированных и блочно-ориентированных устройств различен.

Проблема при отладке драйверов устройств состоит в том, что файл **.EXE** отсутствует, так как для выполнения соответствующих функций драйвер должен быть загружен во время загрузки системы с помощью команды

```
DEVICE = DRIVER.EXT
```

где **EXT** — это расширение **.SYS**, **.COM** или **.BIN**. Это означает, что отлаживаемый драйвер устройства уже резидентен в памяти до начала отладки. Следовательно, функции по выполнению загрузки и перемещения таблицы идентификаторов весьма полезны, поскольку они могут восстановить информацию об идентификаторах для дизассемблированного сегмента памяти (когда драйвер загружен). Как мы увидим далее, команда **File Resident** также очень полезна.

Отладка драйвера устройства

При отладке драйверов устройств можно использовать два подхода. Первый аналогичен отладке TSR, а для второго используются средства удаленной отладки, о которых рассказывается ниже. Для применения этого последнего способа выполните следующие шаги:

- Скомпилируйте драйвер с включенной отладочной информацией.
- С помощью утилиты **TDSTRIP** выделите из драйвера устройства отладочную информацию.
- Скопируйте драйвер устройства на удаленную систему.

- Измените файл **CONFIG.SYS** удаленной системы, чтобы он загружал драйвер удаленной системы. Затем перезагрузите удаленную систему.
- Для получения адреса драйвера загрузите на удаленной системе **TDMEM**.
- Загрузите на удаленной системе **TDREMOTE**.
- Загрузите на локальной системе отладчик, связав его с удаленной системой.
- Загрузите в отладчике с помощью команды **File Symbol Load** таблицу идентификаторов драйвера устройства.
- Используйте команду **File Table Relocate** для помещения первого идентификатора из таблицы идентификаторов в соответствующую ячейку памяти. Таким образом, имеющаяся информация об идентификаторах будет соответствовать вашему коду (программе). Для этого в ответ на подсказку отладчика задайте адрес сегмента **Seg** вашей резидентной программы, который можно определить с помощью **TDMEM**.
- Задайте в начале драйвера устройства точку останова.
- Выберите команду **File Resident**, чтобы сделать резидентным сам отладчик. Это не нарушит резидентности вашего драйвера: когда он будет выполняться в отладчике, он сам станет резидентным при загрузке удаленной системы в результате выполнения файла **CONFIG.SYS**. Единственная резидентной загрузки отладчика заключается в том, что вы можете перейти обратно в DOS и вызвать ваш драйвер устройства.
- Когда вы вернетесь снова к командной строке DOS на удаленной системе, сделайте что-либо для активизации вашего драйвера устройства. Например, выведите информацию на со ответствующее устройство.
- Когда в вашей программе-драйвере встретится точка останова, инициализируется отладчик, а код вашей программы вы ведется в соответствующей точке. Теперь вы можете начать отладку вашей программы. (Кроме того, вы можете повторно войти в отладчик из DOS, дважды нажав клавиши **Ctrl-Break**.)

Удаленная отладка

Удаленная отладка означает с соответствии со своим названием следующее: вы запускаете отладчик на одном компьютере, а отлаживаемую программу — на другом. Две системы могут соединяться через последовательный порт или через локальную сеть LAN, совместимую с **NETBIOS**. Удаленную отладку полезно использовать в следующих ситуациях:

- Вашей программе требуется много памяти, и вы не можете за пускать программу и отладчик на одном компьютере.
- Ваша программа загружается с отладчиком, но для ее правильного функционирования памяти недостаточно. В этом случае в процессе отладки вы будете получать сообщения об ошибках распределения памяти.
- Нужно отладить специальные программы (резидентные программы или драйверы устройств).
- Вы отлаживаете программу Windows.

В случае отладки прикладной программы Windows у вас есть выбор: вы можете либо запустить на одной машине программу и отладчик для Windows (TDW), либо запустить Windows, утилиту **WREMOTE** и прикладную программу на одной машине, а отладчик — на другой.

Требования к программному и аппаратному обеспечению

Для сеанса удаленной отладки вы можете выбрать соединение через последовательный порт или через локальную сеть. В этих случаях используются разные аппаратные средства, однако должны соблюдаться следующие общие требования:

- Рабочая система с памятью, достаточной для загрузки отладчика (локальная система).
- Другой компьютер PC (удаленная система), имеющий достаточный для отлаживаемых программ DOS и **TDREMOTE** объем памяти (или для отлаживаемой программы Windows и **WREMOTE**). Это удаленная система.

Две системы должны соединяться через последовательный порт нуль-модемным кабелем. При соединении через локальную сеть потребуется программное обеспечение, совместимое с Novell

Netware, программное обеспечение, совместимое с Novell Netware (версии IPX и NETBIOS 3.0 или старше).

Запуск сеанса удаленной отладки

Чтобы инициировать сеанс удаленной отладки, подготовке удаленную систему, конфигурируйте и запустите **WREMOTE** (драйвер удаленной отладки), запустите и конфигурируйте на локальной системе TDW и загрузите программу для отладки.

Удаленная система должна содержать следующие файлы: отлаживаемую программу и все необходимые для нее файлы, **WREMOTE.EXE**, **WRSETUP.EXE** (программу конфигурации).

Перед запуском **WREMOTE** с помощью **WRSETUP** нужно задать параметры передачи данных. Для последовательного подключения щелкните «мышью» на кнопке **Serial**, выберите скорость передачи (**Baud Rate**), выберите **Desable Clock Interrupts** и порт. В поле ввода **Starting Directory** введите каталог вашей программы. Если нужно, чтобы **WREMOTE** после завершения отладчика возвращала управление в Windows, установите **Quit When Host Quits**. По умолчанию WREMOTE использует COM1 и скорость 192000 бод.

```
-  WRSetup Turbo Debugger Setup  -

 File    Settings    Help

      -             Remote Driver Settings

                                  OK          Cancel
          Disable clock interrupts
          Quit when TD quits             Baud rate
                                          o 9600
       Starting directory:                * 19200
                                          o 38400
                                          o 115000

         Remote type
          * Serial
          o Network

                                         Comm port
       Network remote name:               * COM1
                                          o COM2
```

При использовании связи через сеть щелкните «мышью» на

кнопке с независимой фиксацией **Network**, в поле ввода **Network Remote Name** задайте имя удаленной системы (по умолчанию **REMOTE**), а в поле **Starting Directory** введите каталог программы. После закрытия окна **WRSETUP** установки сохраняются в файле **TDW.INI**.

После настройки конфигурации **WREMOTE** вы можете загрузить ее, щелкнув «мышью» на пиктограмме **Remote Debugging** или с помощью команды **Windows File Run**. Курсор «мыши» изменяет форму, указывая, что он ждет запуска TDW на другом конце.

Запуск TDW

После запуска на удаленной системе **TDREMOTE** для связи TDW с **TDREMOTE** его нужно правильно конфигурировать. Проще всего это сделать с помощью команды **File Open** (но можно использовать и **Options Misceeellaneous** программы **TDWINST**). В открывающемся диалоговом окне **Load a New Program to Debug** щелкните «мышью» на кнопке **Session**. Открывается окно **Set Session Parameters**. Щелкните «мышью» на кнопке **Serial Remote**. Затем выберите порт (**Remote Link Port**) и скорость передачи (**Link Speed**). Щелкните «мышью» на **OK**. (Порты систем могут быть разными, но скорость должна совпадать.)

Для конфигурации TDW на локальной сети **NETBIOS** запустите на удаленной системе **WREMOTE**, запустите TDW и выберите **File Open**. Открывается окно **Load a New Program**. Чтобы открыть окно **Set Session Parameters** щелкните «мышью» на кнопке **Session**. Выберите кнопку **Network Remote** и задайте имена локальной и удаленной систем (по умолчанию **LOCAL** и **REMOTE**). Затем щелкните на **OK**.

Инициация связи

После настройки TDW для удаленной отладки загрузите программу с помощью диалогового окна **Load a New Program to Debug**. TDW выводит уведомляющее сообщение. После установления связи выводится обычный экран отладчика, и команды его работают так же. Однако вывод программы на экран и ввод с клавиатуры происходит на удаленной системе.

Автоматическая передача файла

После загрузки программы TDW автоматически определяет, нужно ли пересылать программу на удаленную систему. В отношении загрузки программы в удаленную систему отладчик отличается гибкостью. Сначала он проверяет наличие программы на удаленной системе. Если программы там нет, он передает ее. Если программа на удаленной системе имеется, он анализирует дату и время копии программы на локальной системе и удаленной системе. Если копия на локальной системе более поздняя (новая), чем на удаленной, он предполагает, что вы перекомпилировали и перекомпоновали программу и передает ее по линии связи. Учтите однако, что TDW передает только файлы **.EXE**.

Турбо Си ++

Интегрированная среда разработки

TURBO C++ упрощает процесс программирования и делает его более эффективным. При работе в TURBO C++ весь комплекс инструментальных средств, необходимых для написания, редактирования, компиляции, компоновки и отладки программ, оказывается под рукой у пользователя.

Весь этот комплекс возможностей заключен в **Интегрированной Среде Разработки (ИСР)**.

Кроме того, Среда разработки программ TURBO C++ предоставляет следующие дополнительные возможности, которые еще больше упрощают процесс написания программ:

- Возможность отображения на экране монитора значительного числа окон, которые можно перемещать по экрану и размеры которых можно изменять.
- Наличие поддержки «мыши».
- Наличие блоков диалога.
- Наличие команд удаления и вставки (при этом допускается копирование из окна **HELP** и между окнами **EDIT**).
- Возможность быстрого вызова других программ и обратного возврата.
- Наличие в редакторе макроязыка.

ИСР содержит три визуальных компоненты: **строку меню** у верхнего края экрана, **оконную область** в средней части экрана и **строку состояния** у нижнего края экрана. В результате выбора некоторых элементов меню на экран будут выдаваться блоки диалога.

Строка меню и меню

Строка меню представляет собой основное средство доступа ко всем командам меню. Строка меню оказывается невидимой

лишь во время просмотра информации, отображаемой программой и во время перехода к другой программе.

Окна TURBO C++

Большая часть того, что видно и делается в среде TURBO C++, происходит в окне. Окно — это область экрана, которую можно перемещать, размеры которой можно перемещать, изменять, которую можно распахивать на весь экран, ориентировать встык с другими окнами.

В TURBO C++ может существовать произвольное число окон, но в каждый момент активно только одно окно. Активным является то окно, в котором в настоящий момент происходит работа.

Любые вводимые команды или вводимый текст, как правило, относятся только к активному окну.

Существует несколько типов окон, но большая их часть имеет следующие общие элементы:

- строку заголовка;
- маркер закрытия окна;
- полосы прокрутки;
- угол изменения размера окна;
- маркер распахивания окна на весь экран;
- номер окна.

Строка состояния

Строка состояния, расположенная у нижнего края экрана, выполняет следующие функции:

- Напоминает об основных клавишах и клавишах активизации, которые в настоящий момент могут быть применены к активному окну.
- Позволяет установить указатель мыши на обозначения клавиш и кратковременно нажать кнопку мыши для выполнения указанного действия, вместо того, чтобы выбирать команды из меню или нажимать соответствующие клавиши.
- Сообщает, какое действие выполняется программой.

- Предлагает состоящие из одной строки советы и рекомендации по любой выбранной команде меню и элементам блока диалога.

Блоки диалога

Если за элементом меню располагается многоточие, то в результате выбора данной команды будет открыт блок диалога, обеспечивающий удобный способ просмотра и задания многочисленных параметров.

При задании значения в блоке диалога работа происходит с пятью базовыми типами средств управления: указателями выбора, переключателями состояния, кнопками действия, блоками ввода и блоками списка.

Работа с экранным меню

Меню (системное)

Отображается у левого края строки меню. Для вызова следует нажать **ALT+пробел**. При вызове этого меню отображаются команды:

- **About**

При выборе данной команды появляется блок диалога, в котором содержится информация по авторским правам и номер версии TURBO C++. Данное окно закрывается нажатием клавиши **ESC** или **ENTER**.

- **Clear Desktop**

Закрывает все окна и стирает все списки предысторий. Эта команда полезна в тех случаях, когда начинается работа над новым проектом.

- **Repaint Desktop**

Осуществляет регенерацию изображения на экране.

Элементы подменю Transfer

В этом подменю показаны имена всех программ, которые установлены с помощью блока диалога **Transfer**, вызываемого командой **Options/Transfer**. Для запуска программы необходимо выбрать ее имя из системного меню.

Меню File (ALT F)

Это меню позволяет открывать в окнах **EDIT** и создавать исходные файлы программ, сохранять внесенные изменения, выполнять другие действия над файлами, выходить в оболочку DOS и завершать работу с TURBO C++.

Open (F3)

Команда **FILE OPEN** отображает блок диалога, в котором выбирается исходный файл программы, который будет открыт в окне **EDIT**.

Этот блок диалога содержит блок ввода, список файлов, и кнопки **OPEN**, **REPLACE**, **CANCEL** и **HELP**, а также информационную панель.

Здесь можно выполнить одно из действий:

- Ввести полное имя файла и выбрать указатель **REPLACE** или **OPEN**.

В результате выбора **Open** файл загружается в новое окно **Edit**. Для выбора **Replace** должно иметься активное окно **Edit**; в результате выполнения **Replace** содержимое окна заменяется выбранным файлом.

- Ввести имя файла с метасимволами. Это позволяет отфильтровать список файлов в соответствии со спецификацией.
- Выбрать спецификацию файла из списка предыстории, который содержит введенные ранее спецификации файлов.
- Просмотреть содержимое других каталогов, выбрав имя каталога из списка файлов.

Блок ввода позволяет явно ввести имя файла или ввести имя файла с метасимволами **DOS** (***** и **?**). Если ввести имя полностью и нажать **Enter**, Turbo C++ откроет указанный файл. Если ввести имя файла, который система Turbo C++ не может обнаружить, она автоматически создаст и откроет новый файл с таким именем.

Если нажать **?**, когда курсор находится в блоке ввода, то под этим блоком появляется список предыстории, содержащий последние восемь имен файлов, которые были введены ранее.

New

Команда **File New** позволяет открывать новое окно **Edit** со стандартным именем **NONAMExx.C** (где вместо букв **xx** задается число в диапазоне от **00** до **99**).

Файлы с именем **NONAME** используются в качестве временного буфера для редактирования; когда файл с подобным именем сохраняется на диске, Turbo C++ запрашивает действительное имя файла.

Save (F2)

Команда **File Save** записывает на диск файл, находящийся в активном окне **Edit** (если активно окно **Edit** в настоящий момент, если нет, то данным элементом меню нельзя воспользоваться).

Если файл имеет использованное по умолчанию имя **(NONAMEOO.C** и т.п.) TurboC++ откроет блок диалога **Save Editor File**, который позволяет переименовать данный файл и сохранять его в другом каталоге или на другом дисководе.

Save As

Команда **File Save As** позволяет сохранить файл в активном окне **Edit** под другим именем, в другом каталоге или на другом дисководе.

Change Dir

Команда **File Change Dir** позволяет задать идентификатор и имя каталога, которые следует сделать текущими. Текущим является тот каталог, который используется в Turbo C++ для сохранения и поиска файлов. При использовании относительных маршрутов в **Options Directories** они задаются только относительно текущего каталога.

Print

Команда **File Print** печатает содержимое активного окна **Edit** Turbo C++ «раскрывает» символы табуляции (заменяет их соответствующим числом пробелов), а затем посылает файл на устройство печати, заданное в DOS.

Данная команда будет «запрещена», если содержимое активного окна не может быть выведено на печать. Для вывода на печать только выделенного текста следует использовать **Ctrl-K P**.

Get Info

Команда **File Get Info** отображает блок, в котором содержится информация относительно текущего файла.

DOS Shell

Команда **File DOS Shell** позволяет временно выйти из Turbo C++, чтобы выполнить команду DOS или запустить программу. Для возврата в Turbo C++ необходимо ввести с клавиатуры **EXIT** и нажать **Enter**.

Иногда можно обнаружить, что во время отладки не хватает памяти для выполнения этой команды. В этом случае необходимо завершить сеанс отладки командой **Run Program Reset** (**Ctrl-F2**).

Quit (Alt-x)

Команда **File Quit** осуществляет выход из системы Turbo C++, удаляет ее из памяти и передает управление DOS. Если внесены изменения, которые еще не были сохранены, то перед выходом Turbo C++ выдаст запрос на их сохранение.

Значения блока Get Info

Current directory

Имя каталога по умолчанию.

Current file

Имя файла в активном окне.

Extended memory usage

Объем дополнительной памяти, зарезервированной для Turbo C++.

Expanded memory usage

Объем расширенной памяти, зарезервированной для Turbo C++.

Lines compiled

Число откомпилированных строк.

Total warnings

Число выданных системой предупреждающих сообщений.

Totals errors

Число сгенерированных ошибок.

Total time

Время последнего выполнения программы.

Program loaded

Статус отладки.

Program exit

Код возврата от последней завершившейся программы.

Available memory

Объем доступной памяти DOS (640 K).

Last step time

Время выполнения последнего шага отладки.

Меню Edit (Alt-E)

Позволяет выполнять удаления, копирование и вставку текста в окнах **Edit**. Можно также открыть окно текстового буфера для просмотра или редактирования его содержимого. Выбрать текст это значит выделить его цветом:

- Нажать **Shift** с одновременным нажатием стрелки.
- Нажать **Ctrl-K B**, чтобы пометить начало выделяемого блока. Затем переместить курсор в конец фрагмента текста и нажать **Ctrl-K K**.
- Для выбора строки необходимо нажать **Ctrl-K L**. После выделения фрагмента текста становятся доступными команды, из меню **Edit**, и можно использовать текстовый буфер (**Clipboard**). Он взаимодействует с командами меню **Edit**.

Restore Line

Эта команда отменяет действие последней команды редактирования, примененной к какой-либо строке. Она действует только над последней отредактированной строкой.

Cut (Shift-Del)

Удаляет выделенный фрагмент текста из документа и заносит его в текстовый буфер. Затем можно вставить текст в другой документ путем выбора **Paste**.

Copy (Ctrl-Ins)

Эта команда не изменяет выделенный текст, но заносит в текстовый буфер его точную копию. Затем можно вставить текст в другой документ командой **Paste**. Можно скопировать текст из окна **Help**; следует использовать **Shift** и клавиши управления курсором.

Paste (Shift-Ins)

Эта команда вставляет текст из текстового буфера в текущее окно в позиции курсора.

Show Clipboard

Эта команда открывает окно **Clipboard**, в котором хранятся фрагменты текста, удаленного и скопированного из других окон.

Clear (Ctrl-Del)

Эта команда удаляет выбранный фрагмент текста, но не заносит его в текстовый буфер. Это означает, что восстановить удаленный текст нельзя.

Меню Search (Alt-S)

Меню **Search** выполняет поиск текста, объявлений функций, а также местоположение ошибок в файлах.

Search Find

Команда **Search Find** отображает блок диалога **Find**, который позволяет ввести образец поиска и задать параметры, влияющие на процесс поиска.

Эта команда может быть также вызвана с помощью **Ctrl-Q-F**.

Replace (Ctrl Q A)

Команда **Search Replace** отображает блок диалога для ввода искомого текста и текста, на который его следует заменить.

Search Again (Ctrl L)

Команда **Search Again** повторяет действие последней команды **Find** или **Replace**. Все параметры, которые были заданы при последнем обращении к использованному блоку диалога (**Find** или **Replace**), остаются действительными при выборе данной команды.

Меню Run (Alt-R)

Команды этого меню выполняют программу, а также инициализируют и завершают сеанс отладки.

Run (Ctrl-F9)

Команда **Run** выполняет программу, используя те аргументы, которые переданы программе с помощью команды **Run Arguments**.

Trace Into (F7)

Эта команда выполняет программу по операторам. По достижению вызова функции будет выполняться каждый ее оператор вместо того, чтобы выполнить эту функцию за один шаг. Этой командой следует пользоваться для перемещения выполнения в функцию, которая вызывается из отлаживаемой функции.

Program Reset (Ctrl-F2)

Команда **Run Program Reset** прекращает текущий сеанс отладки, освобождает память программы и закрывает все открытые файлы, которые использовались в программе.

Over

Команда **Run Step Over** выполняет следующий оператор в текущей функции без вхождения в функции более низкого уровня, даже если эти функции доступны отладчику.

Командой **Step Over** следует пользоваться в случаях, когда необходимо отладить функцию в пооператорном режиме выполнения без вхождения в другие функции.

Arguments

Команда **Run Arguments** позволяет задать выполняемой программе аргументы командной строки точно так же, как если бы они вводились в командной строке DOS. Команды переназначения ввода/вывода DOS будут игнорироваться.

Меню Compile (C)

Команды из меню **Compile** используются для компиляции программы в активном окне, а также для полной или избирательной компиляции проекта.

EXE File

Команда **Compile Make EXE File** вызывает Менеджер проектов для создания **EXE**-файла.

Link EXE File (только при полном наборе меню)

Команда **Compile Link EXE File** использует текущие **OBJ** и **LIB**-файлы и компонует их, не производя избирательной компиляции.

Меню Debug (Alt F9)

Команды меню **Debug** управляют всеми возможностями интегрированного отладчика.

Inspect (Alt F4)

Команда **Debug Inspect** открывает окно **Inspector**, которому позволяет проанализировать и модифицировать значения элемента данных.

Меню Options (Alt-O)

Меню **Options** содержит команды, которые позволяют просматривать и модифицировать стандартные параметры, определяющие функционирование Turbo C++.

Структура файла, типы данных и операторов ввода-вывода

Функция Main

Каждый исполняемый файл системы Турбо С++ (программа) должен содержать функцию **main**.

Код, задающий тело функции **main**, заключается в фигурные скобки **{и}**.

Общая структура функции **main** такова:

```
main()
{
/* Код, реализующий main */
}
```

Комментарии

Текст на Турбо С++, заключенный в скобки /* и */, компилятором игнорируется.

Комментарии служат двум целям: документировать код и облегчить отладку. Если программа работает не так, как надо, то иногда оказывается полезным закомментировать часть кода (т.е. вынести ее в комментарий), заново скомпилировать программу и выполнить ее.

Если после этого программа начнет работать правильно, то значит, закомментированный код содержит ошибку и должен быть исправлен.

Директивы Include

Во многие программы на Турбо C++ подставляются один или несколько файлов, часто в самое начало кода главной функции **main**.

Появление директив

```
#include <файл_1>
#include "файл_2"
...
#include <файл_n>
```

приводит к тому, что препроцессор подставляет на место этих директив тексты файлов **файл_1**, **файл_2** , **...** , **файл_n** соответственно.

Если имя файла заключено в угловые скобки **<...>**, то поиск файла производится в специальном разделе подстановочных файлов. В отличие от многих других операторов Турбо C++ директива **Include** не должна оканчиваться точкой с запятой.

Макро

С помощью директивы **#define**, вслед за которой пишутся имя макро и значение макро, оказывается возможным указать препроцессору, чтобы он при любом появлении в исходном файле на Турбо C++ данного имени макро заменял это имя на соответствующие значения макро.

Например, директива

```
#define pi 3.1415926
```

связывает идентификатор **pi** со значением **3.1415926**. После значения макро **(;)** не ставится.

Типы данных

В Турбо C++ переменные должны быть описаны, а их тип специфицирован до того, как эти переменные будут использованы.

При описании переменных применяется префиксная запись, при которой вначале указывается тип, а затем — имя переменной.

Например:

```
float weight;
int exam_score;
```

```
char ch;
```

С типом данных связываются и набор предопределенных значений, и набор операций, которые можно выполнять над переменной данного типа.

Переменные можно инициализировать в месте их описаний.

Пример:

```
int height = 71 ;
float income =26034.12 ;
```

Простейшими скалярными типами, предопределёнными в Турбо С++, являются

- **char** — представляется как однобайтовое целое число
- **int** — двубайтовое целое
- **long** — четырёхбайтовое целое
- **float** — четырёхбайтовое вещественное
- **double** — восьмибайтовое вещественное

Оператор printf: вывод на терминал

Функцию **printf** можно использовать для вывода любой комбинации символов, целых и вещественных чисел, строк, беззнаковых целых, длинных целых и беззнаковых длинных целых.

Пример:

```
printf("\nВозраст Эрика - %d. Его доход $%.2f",age,income);
```

Предполагается, что целой переменной **age** (возраст) и вещественной переменной **income** (доход) присвоены какие-то значения.

Последовательность символов **«\n»** переводит курсор на новую строку.

Последовательность символов **«Возраст Эрика»** будет выведена с начала новой строки. Символы **%d —** это спецификация для целой переменной **age**.

Следующая литерная строка «**Его доход $**». **%2f —** это спецификация (символ преобразования формата) для вещественного значения, а также указание формата для вывода только двух цифр после десятичной точки. Так выводится значение переменной **income**.

Символ формата	Тип выводимого объекта
%c	**char**
%s	строка
%d	**int**
%o	**int** (в восьмеричном виде)
%u	**unsigned int**
%x	**int** (в шестнадцатеричном виде)
%ld	**long** (в десятичном виде)
%lo	**long** (в восьмеричном виде)
%lu	**unsigned long**
%lx	**long** (в шестнадцатеричном виде)
%f	**float**/double (с фиксированной точкой)
%e	**float**/double (в экспоненциальной форме)
%g	**float**/double (в виде **f** или **e** в зависимости от значения)
%lf	**long float** (с фиксированной точкой)
%le	**long float** (в экспоненциальной форме)
%lg	**long float** (в виде **f** или **e** в зависимости от значения)

Оператор scanf: ввод с клавиатуры

Оператор **scanf** является одной из многих функций ввода, имеющихся во внешних библиотеках.

Каждой вводимой переменной в строке функции **scanf** должна соответствовать спецификация. Перед именами переменных необходимо оставить символ **&**. Этот символ означает **«взять адрес»**.

Пример:

```
#include<stdio.h>
main()
{
int weight, /*вес*/ height; /*рост*/
printf(" Введите ваш вес: ");
scanf("%d", &weight);
```

```
printf(" Введите ваш рост: ");
scanf("%d", &height);

printf("\n\nВес = %d, рост = %d\n",
weight,height);
}
```

Арифметические, логические операции и операции отношения и присваивания

Основу языка Турбо С++ составляют операторы. Оператором выражения называют выражение, вслед за которым стоит точка с запятой. В Турбо С++ точки с запятой используются для разделения операторов. Принято группировать все операторы в следующие классы:

- присваивания,
- вызов функции,
- ветвления,
- цикла.

В операторе присваивания используется операция присваивания **=.**

Например:

```
c = a * b;
```

Действие такого оператора можно описать следующими словами: «с присваивается значение **a**, умножение на **b**». Значение, присваиваемое переменной **c**, равняется произведению текущих значений переменных **a** и **b**.

Операторы часто относятся более чем к одному из четырех классов.

Например, оператор

```
if ( ( c = cube( a * b ) ) > d )
...
```

составлен из представителей следующих классов: присваивания, вызов функции, и ветвление.

К понятию оператора вплотную примыкает понятие **операции**.

Различают следующие группы операций Турбо C++:

- арифметические операции
- операции отношения
- операции присваивания
- логические операции
- побитовые операции
- операция вычисления размера (sizeof)
- операция следования (запятая).

Арифметические операции

К арифметическим операциям относятся:

- сложение (+)
- вычитание (-)
- деление (/)
- умножение (*)
- остаток (%).

Все операции (за исключением остатка) определены для переменных типа **int**, **char**, **float**. Остаток не определен для переменных типа **float**. Все арифметические операции с плавающей точкой производятся над операндами двойной точности.

Операции отношения

В языке определены следующие операции отношения:

- проверка на равенство (==)
- проверка на неравенство (!=)
- меньше (<)
- меньше или равно (<=)
- больше (>)
- больше или равно (>=).

Все перечисленные операции вырабатывают результат типа **int**. Если данное отношение между операндами истинно, то значение этого целого — единица, а если ложно, то нуль.

Все операции типа больше-меньше имеют равный приоритет, причем он выше, чем приоритет операций **==** и **!=**. Приоритет операции присваивания ниже приоритета всех операций отношений. Для задания правильного порядка вычислений используются скобки.

Логические операции

В языке имеются три логические операции:

- && операции И (and)
- || операции ИЛИ (or)
- ! отрицание

Аргументами логических операций могут быть любые числа, включая задаваемые аргументами типа **char**. Результат логической операции — единица, если истина, и нуль, если ложь. Вообще все значения, отличные от нуля, интерпретируются как истинные.

Логические операции имеют низкий приоритет, и поэтому в выражениях с такими операциями скобки используются редко.

Вычисление выражений, содержащих логические операции, производится слева направо и прекращается (усекается), как только удается определить результат.

Если выражение составлено из логических утверждений (т.е. выражения, вырабатывающие значения типа **int**), соединенных между собой операцией **И (&&)**, то вычисление выражения прекращается, как только хотя бы в одном логическом утверждении вырабатывается значение нуль.

Если выражение составлено из логических утверждений, соединенных между собой операцией **ИЛИ** (||), то вычисление выражения прекращается, как только хотя бы в одном логическом утверждении вырабатывается ненулевое значение.

Вот несколько примеров, в которых используются логические операции:

```
if( i > 50 && j == 24)
    ...
    if( value1 < value2 && (value3 > 50 || value4 < 20) )
    ...
```

Операции присваивания

К операциям присваивания относятся **=, +=, -=, *=** и **/=,** а также **префиксные** и **постфиксные** операции **++** и **--**.

Все операции присваивания присваивают переменной результат вычисления выражения. Если тип левой части присваивания отличается от типа правой части, то тип правой части приводится к типу левой.

В одном операторе операция присваивания может встречаться несколько раз. Вычисления производятся справа налево.

Например:

```
a = ( b = c ) * d;
```

Вначале переменной **d** присваивается значение **c**, затем выполняется операция умножения на **d**, и результат присваивается переменной **a**.

Операции **+=, -=, *=** и **/=** являются укороченной формой записи операции присваивания. Их применение проиллюстрируем при помощи следующего описания:

```
a += b означает a = a + b
a -= b означает a = a - b
a *= b означает a = a * b
a /= b означает a = a / b
```

Префиксные и постфиксные операции **++** и **--** используют для увеличения (инкремент) и уменьшения (декремент) на единицу значения переменной.

Семантика указанных операций следующая:

- **++a** — увеличивает значение переменной **a** на единицу до использования этой переменной в выражении.
- **a++** — увеличивает значение переменной **a** на единицу после использования этой переменной в выражении.
- **--a** — уменьшает значение переменной **a** на единицу до использования этой переменной в выражении.
- **a--** — уменьшает значение переменной **a** на единицу после использования этой переменной в выражении.

Операцию **sizeof** (размер) можно применить к константе, типу или переменной. В результате будет получено число байтов, занимаемых операндом.

Например:

```
printf ( "\nРазмер памяти под целое %d", sizeof( int) );
printf ( "\nРазмер памяти под символ %d", sizeof( char) );
```

Логическая организация программы и простейшее использование функций

Процесс разработки программного обеспечения предполагает разделение сложной задачи на набор более простых задач и заданий. В Турбо С++ поддерживаются функции как логические единицы (блоки текста программы), служащие для выполнения конкретного задания. Важным аспектом разработки программного обеспечения является функциональная декомпозиция.

Функции имеют нуль или более формальных параметров и возвращают значение скалярного типа, типа **void** (пусто) или указатель. При вызове функции значения, задаваемые на входе, должны соответствовать числу и типу формальных параметров в описании функции.

Если функция не возвращает значения (т.е. возвращает **void**), то она служит для того, чтобы изменять свои параметры (вызывать побочный эффект) или глобальные для функции переменные.

Например, функция, возвращающая куб ее вещественного аргумента:

```
double cube( double x )
{
return x * x * x ;
}
```

Аргумент **x** типа **double** специфицируется вслед за первой открывающей скобкой. Описание **extern**, помещаемое в функцию **main**, является ссылкой вперед, позволяющей использовать функцию **cube** в функции **main**. Ключевое слово **extern** можно опускать, но сама ссылка вперед на описание функции является обязательной.

Логическая организация простой программы

Турбо С++ предоставляет необычайно высокую гибкость для физической организации программы или программной системы.

Структура каждой функции совпадает со структурой главной программы (**main**). Поэтому функции иногда еще называют подпрограммами.

Подпрограммы решают небольшую и специфическую часть общей задачи.

Использование констант различных типов

В языке Турбо С++ имеются четыре типа констант:

- целые
- вещественные (с плавающей точкой)
- символьные
- строковые.

Константы целого типа

Константы целого типа могут задаваться в десятичной, двоичной, восьмеричной или шестнадцатеричной системах счисления.

Десятичные целые константы образуются из цифр. Первой цифрой не должен быть нуль.

Восьмеричные константы всегда начинаются с цифры нуль, вслед за которой либо не стоит ни одной цифры, либо стоят несколько цифр от нуля до семерки.

Шестнадцатеричные константы всегда начинаются с цифры нуль и символа **x** или **X**, все, за которыми может стоять одна или более шестнадцатеричных цифр.

Шестнадцатеричные цифры — это десятичные цифры от **0** до **9** и латинские буквы: **a, b, c, d, e, f**, или **A, B, C, D, E, F**.

Например: задание константы **3478** в десятичном, восьмеричном и шестнадцатеричном виде:

```
int a = 3478,
b = 06626,
```

```
c = 0xD96;
```

К любой целой константе можно справа приписать символ **l** или **L**, и это будет означать, что константа — длинная целая (**long integer**). Символ **u** или **U**, приписанный к константе справа, указывает на то, что константа целая без знака (**unsigned long**).

Считается, что значение любой целой константы всегда неотрицательно. Если константе предшествует знак минус, то он трактуется как операция смены знака, а не как часть константы.

Константы вещественного типа

Константы с плавающей точкой (называемые вещественными) состоят из цифр, десятичной точки и знаков десятичного порядка **e** или **E**.

```
1.      2e1    .1234      .1e3
.1      2E1    1.234      0.0035e-6
1.0     2e-1   2.1e-12    .234
```

Символьные константы

Символьные константы заключаются в апострофы (кавычки). Все символьные константы имеют в Турбо С++ значение типа **int** (целое), совпадающее с кодом символа в кодировке **ASCII**.

Одни символьные константы соответствуют символам, которые можно вывести на печать, другие — управляющим символам, задаваемым с помощью **esc**-последовательности, третьи — форматирующими символами, также задаваемым с помощью **esc**-последовательности.

Например:

- символ «апостроф» задается как '\''
- переход на новую строку — как '\'
- обратный слэш — как '\\'

Каждая **esc**-последовательность должна быть заключена в кавычки.

Управляющие коды

- **\n** — Новая строка
- **\t** — Горизонтальная табуляция
- **\v** — Вертикальная табуляция

- **\b** — Возврат на символ
- **\r** — Возврат в начало строки
- **\f** — Прогон бумаги до конца страницы
- \\ — Обратный слэш
- \' — Одинарная кавычка
- \" — Двойная кавычка
- **\a** — Звуковой сигнал
- **\?** — Знал вопроса
- **\ddd** — Код символа в **ASCII** от одной до трех восьмеричных цифр
- **\xhhh** — Код символа в **ASCII** от одной до трех шестнадцатеричных цифр.

Строковые константы

Строковые константы состоят из нуля или более символов, заключенных в двойные кавычки. В строковых константах управляющие коды задаются с помощью **esc**-последовательности. Обратный слэш используется как символ переноса текста на новую строку.

Пример описания строковых констант:

```
# include <stdio.h>
main( )
{
char *str1, *str2;
str1=" Пример использования\n\n";
str2="строковых\
констант.\n\n";
printf(str1);
printf(str2);
}
```

Управляющие структуры

Управляющие структуры или операторы управления служат для управления последовательностью вычислений в программе. Операторы ветвления и циклы позволяют переходить к выполнению другой части программы или выполнять какую-то

часть программы многократно, пока удовлетворяется одно или более условий.

Блоки и составные операторы

Любая последовательность операторов, заключенная в фигурные скобки, является составным оператором (блоком). Составной оператор не должен заканчиваться **(;)**, поскольку ограничителем блока служит сама закрывающаяся скобка. Внутри блока каждый оператор должен ограничиваться **(;)**.

Составной оператор может использоваться везде, где синтаксис языка допускает применение обычного оператора.

Пустой оператор

Пустой оператор представляется символом **(;)**, перед которым нет выражения. Пустой оператор используют там, где синтаксис языка требует присутствия в данном месте программы оператора, однако по логике программы оператор должен отсутствовать.

Необходимость в использовании пустого оператора часто возникает, когда действия, которые могут быть выполнены в теле цикла, целиком помещаются в заголовке цикла.

Операторы ветвления

К операторам ветвления относятся **if**, **if else**, **?**, **switch** и **go to**. Общий вид операторов ветвления следующий:

```
if (логическое выражение)
оператор;
----------------------------------------
if (логическое выражение)
оператор_1;
else
оператор_2;
----------------------------------------
<логическое выражение> ? <выражение_1> : <выражение_2>;
Если значение логического выражения истинно, то вычисляется
выражение_1, в противном случае вычисляется выражение_2.
----------------------------------------
switch (выражение целого типа)
{
case значение_1:
```

```
последовательность_операторов_1;
break;
case значение_2:
последовательность_операторов_2;
break;
. . .
case значение_n:
последовательность_операторов_n;
break;
default:
последовательность_операторов_n+1;
}
```

Ветку **default** можно не описывать. Она выполняется, если ни одно из вышестоящих выражений не удовлетворено.

Оператор цикла

В Турбо С++ имеются следующие конструкции, позволяющие программировать циклы: **while**, **do while** и **for**. Их структуру можно описать следующим образом:

```
while( логическое выражение)
   оператор;
Цикл с проверкой условия наверху
---------------------------
do
   оператор;
while (логическое выражение);
Цикл с проверкой условия внизу
---------------------------
for (инициализация, проверка, новое_значение)
оператор;
   --------------------------
```

Приемы объявления и обращения к массивам, использование функций и директивы define при работе с массивами

Массивы — это набор объектов одинакового типа, доступ к которым осуществляется прямо по индексу в массиве. Обращение к массивам в Турбо С++ осуществляется и с помощью указателей.

Массивы можно описывать следующим образом:

```
тип_данных имя_массива [размер массива];
```

Используя имя массива и индекс, можно адресоваться к элементам массива:

```
имя_массива [значение индекса]
```

Значения индекса должны лежать в диапазоне от нуля до величины, на единицу меньшей, чем размер массива, указанный при его описании.

Вот несколько примеров описания массивов:

```
char name [ 20 ];
int grades [ 125 ];
float income [ 30 ];
double measurements [ 1500 ];
```

Первый из массивов (**name**) содержит 20 символов.

Обращением к элементам массива может быть **name [0]**, **name [1]**, **...**, **name [19]**.

Второй массив (**grades**) содержит 125 целых чисел. Обращением к элементам массива может быть **grades [0]**, **grades [1]**, **...**, **grades [124]**.

Третий массив (**incom**) содержит 30 вещественных чисел. Обращением к элементам массива может быть **income [0]**, **incom [1]**, **...**, **income [29]**.

Четвертый массив (**measurements**) содержит 1500 вещественных чисел с двойной точностью. Обращением к элементам массива может быть **measurements [0]**, **measurements [1]**, **...**, **measurements [1499]**.

Вот программа, иллюстрирующая использование массивов (Файл **array.c**):

```
#include <stdio.h>
#define size 1000
int data [size];
main ( )
{
extern float average (int a[],
int s );
int i;
for ( i=0; i<size ; i++_)
```

```
data [ i ]= i;
printf ( "\nСреднее значение массива data =%f\n",average
(data,size));
}
float average (int a[ ] ,int s )
{
 float sum=0.0;
 int i;
 for ( i=0; i<s ; i ++)
 sum+=a[ i ];
 return sum/s;
}
```

В программе заводится массив на 1000 целых чисел. При помощи функции **average** подсчитывается сумма элементов этого массива.

Первым формальным параметром функции **average** является массив. В качестве второго параметра функции передается число суммируемых значений в массиве **a**.

Обратите внимание на использование константы **size** (размер). Если изменяется размерность массива, задаваемая этой константой, то это не приводит к необходимости менять что-либо в самом коде программы.

Трюки программирования

Правило «право-лево»

Существенный принцип анализа сложных синтаксических конструкций языка, вроде «указатель на функцию, возвращающую указатель на массив из трёх указателей на функции, возвращающие значение **int**» чётко формализован в виде правила «право-лево». Всё предельно просто. Имеем:

- **()** — функция, возвращающая...
- **[]** — массив из...
- ***** — указатель на...

Первым делом находим имя, от которого и будем плясать.

Следующий шаг — шаг вправо. Что там у нас справа? Если **()**, то говорим, что «Имя есть функция, возвращающая...». (Если между скобок что-то есть, то «Имя есть функция, принимающая то, что между скобок, и возвращающая...»). Если там **[]**, то «Имя есть массив из...». И подобным вот образом мы идём вправо до тех пор, пока не дойдём до конца объявления или правой «**)**» скобки. Тут тормозим...

...и начинаем танцевать влево. Что у нас слева? Если это что-то не из приведенного выше (то есть не **()**, **[]**, *****), то попросту добавляем к уже существующей расшифровке. Если же там что-то из этих трёх символов, то добавляем то, что написано выше. И так танцуем до тех пор, пока не дотанцуем до конца (точнее — начала объявления) или левой «**(**» скобки. Если дошли до начала, то всё готово. А если дошли до «**(**», то по уже означенной итеративности переходим к шагу «Пляски вправо» и продолжаем.

Пример:

```
int (*(*(*fptr)())[3])();
```

Находим имя и записываем «**fptr есть...**».

Шаг вправо, но там «**)**», потому идём влево:

```
int (*(*(*fun)())[3])();
```

и получаем «**fptr есть указатель на...**».

Продолжаем ехать влево, но тут «**(**». Идём вправо:

```
int (*(*(*fun)())[3])();
```

получаем «**fptr есть указатель на функцию, возвращающую...**» Снова «**)**», опять влево. Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на...**» Слева опять «**(**», идём вправо. Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх...**» И снова справа «**)**», отправляемся влево. Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх указателей на...**» Снова разворот вправо по причине «**(**». Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх указателей на функции, возвращающие...**» Тут конец описания, поехали влево и получили окончательную расшифровку:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх указателей на функции, возвращающие int**».

Именно то, чего мы хотели.

STLport 4.0

В июле 2000 года наконец-то вышла новая версия библиотеки STLport 4.0. Для тех, кто еще не в курсе, что это такое, объясним: это свободно распространяемая реализация стандартной библиотеки шаблонов для множества различных компиляторов и операционных систем. Помимо всего прочего, STLport доступен не только для современных компиляторов, более или менее удовлетворяющих стандарту языка, но и для некоторых старых компиляторов, в частности Borland C++ 5.02 или MS Visual C++ 4.0.

Четвертая версия STLport отличается от предыдущей главным образом тем, что теперь в нее входит полная поддержка потоков (ранее приходилось использовать потоки из библиотеки, поставляемой с конкретным компилятором). Реализация потоков взята из SGI (как, впрочем, и весь STLport). Вообще, STLport начал развиваться как попытка перенести известную библиотеку SGI STL на gcc и sun cc. Таким образом, с выходом четвертой версии, STLport стал полноценной библиотекой, соответствующей стандарту языка, во всяком случае, у него появились претензии на это.

Понятно, что применение одной и той же библиотеки на разных платформах, это уже большой плюс — потому что никогда точно заранее не известно, что, где и как будет плохо себя вести. Можно только лишь гарантировать, что программа, при переносе с одного компилятора на другой, все-таки будет себя плохо вести даже в том случае, если скомпилируется. Использование одной библиотеки шаблонов очень сильно увеличивает шансы на то, что не будет проблем тогда, когда программист увидит отсутствие в STL нового компилятора какого-нибудь контейнера. К примеру, в g++-stl-3 нет std::wstring. То есть, шаблон std::basic_string есть, и std::string является его инстанционированием на char, но попытка подставить туда же wchar_t ни к чему хорошему не приведет (в частности, из-за того, что в методе c_str() есть исключительная строчка вида return "").

Но и кроме единых исходных текстов у STLport есть еще несколько интересных возможностей и особенностей. Во-первых, это **debug mode**, при котором проверяются все условия, которые только возможны. В частности, в этом режиме при попытке работать с неинициализированным итератором будет выдано соответствующее ругательство. Согласитесь, это удобно.

Во-вторых, в STLport есть несколько нестандартных контейнеров, таких как **hash_map**, например. Зачем? Ну, в силу того что стандартный **map** как правило реализован на сбалансированных деревьях поиска (как более общий способ обеспечения быстрого поиска при разнородных данных), и что делать в том случае, когда все-таки известна хорошая хеш-функция для определенных элементов, не особенно понятно (ну, за исключением того, чтобы написать подобный контейнер самостоятельно).

В третьих, поддержка многопоточности. То есть, STLport можно безопасно использовать в программах, у которых более одного потока выполнения. Это досталось STLport еще от SGI STL, в которой очень много внимания уделялось именно безопасности использования.

Помимо того, если вдруг возникли какие-то проблемы с STL, то можно попытаться взять STLport — быть может, проблем станет поменьше.

Новый язык программирования от Microsoft: C#

Фирма Microsoft создала новый язык программирования, сделанный на основе Си и C++ и Java, который она назвала C# (C sharp).

Чтобы реально посмотреть язык программирования, возьмем программу «Hello, world!» из C# Language Reference:

```
using System;
class Hello
{
    static void Main() {
        Console.WriteLine("Hello, world");
    }
}
```

Это очень сильно напоминает Java. Таким образом, что имеется в наличии:

- Убрали селектор **->**, впрочем это, возможно и правильно: и «точка» и «стрелка» выполняют, в принципе, одни и те же функции с точки зрения ООП, так что в этом есть намеки на концептуальность. В принципе, это стало вероятным благодаря тому, что в C# есть типы «значения» (такие как, **int**, **char**, структуры и перечисления) и типы «ссылки», которыми являются объекты классов, массивы.
- Точно так же, как и в Java, перенесли метод **main** внутрь класса.
- Точно так же, как и в Java, в программах на C# теперь нет необходимости в декларациях без дефиниций, т.е. компилятор многопроходный.

- Конечно же, не смогли обойтись без автоматического сборщика мусора, так что в C#, так же как и в Java, не требуется беспокоиться об удалении памяти из-под объектов. Тем не менее, введена такая возможность, под названием «**unsafe code**», используя которую можно работать с указателями напрямую.
- Появился тип **object** с понятными последствиями: все типы (включая типы «значения») являются потомками **object**.
- Между **bool** и **integer** нет кастинга по умолчанию. Тип **char** — это **Unicode** символ (так же, как и в Java).
- Есть поддержка настоящих многомерных массивов (а не массивов массивов).

В отличие от Java, в C# выжил оператор **goto**.

Появился оригинальный оператор **foreach**:

```
static void WriteList(ArrayList list) {
 foreach (object o in list)
  Console.WriteLine(o);
}
```

который позволяет обойти контейнер.

Есть еще два интересных оператора: **checked** и **unchecked**. Они позволяют выполнять арифметические операции с проверкой на переполнение и без него.

Существует поддержка многопоточности при помощи оператора **lock**.

Отсутствует множественное наследование — вместо него, как и в Java, добавлена поддержка интерфейсов. Кстати сказать, структуры теперь совсем не тоже самое, что и классы. В частности, структуры не могут быть наследованы.

Добавлена поддержка свойств (property).

На языковом уровне введена поддержка отклика на события.

Введены определяемые пользователями атрибуты для поддержки систем автодокументации.

Кардинальное отличие от Java — наличие компилятора в машинный код. То есть, можно предположить, что программы на

C# будут выполняться несколько быстрее, чем написанные на Java.

Вообще, можно говорить о том, что Microsoft учла традиционные нарекания в сторону Java в своем новом языке. В частности, оставлена от C++ перегрузка операторов.

Компания Microsoft утверждает, что создала язык для написания переносимых web-приложений и старается всячески показать свою собственную активность в этом направлении. В частности, компания Microsoft направила запрос на стандартизацию C#.

В принципе, ясно, зачем все это нужно. Компании Microsoft, несомненно, понадобился свой язык программирования такого же класса, как и Java. Пускать же Java к себе в Microsoft никто не намеревался, вот и получился C#. Понятно, что в данном случае язык программирования сам по себе представляет достаточно малую ценность, в силу того что Java хороша своей переносимостью, а переносимость ей гарантирует мощная и обширная стандартная библиотека, употребляя которую нет надобности вызывать какие-то системно- или аппаратно-зависимые фрагменты кода. Поэтому на текущий момент ничего определенного сказать о судьбе C# нельзя — хотя бы потому, что у него пока что нет подобной библиотеки.

Все же, в ближайшие несколько лет будет очень интересно следить за развитием C# и Java. В принципе, еще недавно представлялось, что уже невозможно вытеснить Java из своей ниши инструмента для относительно простого создания переносимых приложений, но вот, Microsoft решилась на эту попытку. Учитывая то, что в свое время было очевидно главенство Netscape на рынке броузеров, ожидать можно всего.

C++ Builder

В первую очередь оговоримся, что здесь мы будем рассматривать C++ Builder именно как «builder», т.е. программный инструмент класса RAD (Rapid Application Development, быстрое создание приложений) и, в общем-то, большая часть здесь написанного в одинаковой степени применимо ко всем подобным средствам.

Итак, C++ Builder облегчает процесс создания программ для ОС Windows с графическим интерфейсом пользователя. При его помощи одинаково просто создать диалог с тремя кнопочками «Yes», «No», «Cancel» или окно текстового WYSIWYG редактора с возможностью выбора шрифтов, форматирования, работы с файлами формата rtf. При этом C++ Builder автоматически создает исходный текст для пользовательского интерфейса: создает новые классы, объекты, добавляет необходимые переменные и функции. После всего этого «рисование» пользовательского интерфейса превращается, буквально, в наслаждение для эстетов: сюда добавим градиент, здесь цвет изменим, тут шрифт поменяем, а сюда мы поместим картинку.

После того, как вся эта красота набросана, наступает менее привлекательная работа — написание функциональной части. Тут C++ Builder ничем помочь не может, все приходиться делать по старинке, позабыв про «манипулятор мышь» и касаясь исключительно клавиатуры.

Итог?.. Как обычно: красота неписанная на экране. Этих программ, которые рисовали эстетствующие программисты в настоящее время видимо-невидимо, ими можно любоваться, распечатывать картинки с экрана и делать из них художественные галереи...

Что же тут плохого? Ничего, если не считать того, что при таком подходе к программированию создание программного продукта начинает идти не от его «внутренностей» (функционального наполнения), а от пользовательского интерфейса и в итоге получается, если «наполнение» достаточно сложное (сложнее передачи текста от одного элемента пользовательского интерфейса другому), то оно становится не только системнозависимым, но и компиляторозависимым, что уж абсолютно неприятно.

Кроме того, простота «рисования» пользовательского интерфейса, а, вернее, ненаказуемость (например, объемным программированием) использования всевозможных сложных компонентов скрывает в себе некоторые опасности. Связано это с тем, что создание удобного пользовательского интерфейса это задача сама по себе довольно трудная и требующая особенного образования. При этом всякий уважающий себя программист

всегда уверен в том, что уж он-то точно сможет сделать пользовательский интерфейс предельно удобным и красивым.

Почему настолько разительно отличаются домашние страницы и страницы профессиональных web-дизайнеров? Вследствие того что последние имеют очень много узкоспециализированных знаний о восприятии человеком информации на экране монитора и, благодаря этому, могут разместить информацию не «красиво», а так, как это будет удобным. То же самое и с пользовательским интерфейсом — вопрос о том, как должна выглядеть конкретная кнопка и в каком месте она должна находиться не так прост, как кажется. Вообще, дизайнер пользовательского интерфейса это совершенно исключительная профессия, которая, к сожалению, у нас еще не распространена.

Тот факт, что функциональное наполнение становится зависимым от используемой библиотеки пользовательского интерфейса, просто абсурден. Подставьте в предыдущее предложение взамен «функционального наполнения» конкретный продукт, и вы уясните о чем мы хотим сказать: «расчет химических реакций», «анализ текста» и т.д.

Помимо того, сама библиотека пользовательского интерфейса в C++ Builder довольно оригинальна. Это VCL (Visual Component Library), всецело позаимствованная из Delphi, т.е. написанная на Паскале. По Паскалевским исходникам автоматически создаются заголовочные файлы, которые в дальнейшем включаются в файлы, написанные на C++. Необходимо сказать, что классы, которые представляют из себя VCL-компоненты это не обычные C++ классы; для совместимости с Delphi их пришлось отчасти изменить (скажем, VCL-классы не могут участвовать во множественном наследовании); т.е. в C++ Builder есть два вида классов: обычные C++ классы и VCL-классы.

Помимо всего прочего, C++ Builder еще и вреден. Вследствие того что очень много начинающих программистов используют его, расхваливают за то, что при его помощи все так просто делается и не подозревают о том, что это, на самом деле, не правильно. Ведь область применения C++ Builder, в общем-то, достаточно хорошо определена — это клиентские части для каких-либо БД. В нем все есть для этого: быстрое создание

интерфейса, генераторы отчетов, средства сопряжения с таблицами. Но все, что выходит за границы данной области, извините, надо писать «как обычно».

Связано это с тем, что, создание программ, которые в принципе не переносимы — это просто издевательство над идеями C++. Ясно, что написать программу, которая компилируется несколькими компиляторами это в принципе сложно, но сделать так, чтобы это было ко всему прочему и невозможно, до чрезвычайности неприлично. Всякая программа уже должна изначально (и это даже не вопрос для обсуждения) иметь очень отчетливую грань между своим «содержанием» и «пользовательским интерфейсом», между которыми должна быть некоторая прослойка (программный интерфейс) при помощи которой «пользовательский интерфейс» общается с «содержанием». В подобном виде можно сделать хоть десяток пользовательских интерфейсов на различных платформах, очень просто «прикрутить» COM или CORBA, написать соответствующий этой же программе CGI скрипт и т.д. В общем, немало достоинств по сравнению с жестким внедрением библиотеки пользовательского интерфейса внутрь программы против одного преимущества обратного подхода: отсутствие необходимости думать перед тем, как начать программировать.

Необходимо сказать, что C++ Builder или Delphi такой популярности как у нас, за границей не имеют. Там эту же нишу прочно занял Visual Basic, что достаточно точно говорит об области применения RAD-средств.

C++ Builder буквально навязывает программисту свой собственный стиль программирования, при котором, даже при особом желании, перейти с C++ Builder на что-то другое уже не предоставляется возможным.

Помимо того, быстрое создание интерфейса это еще не панацея от всех бед, а, скорее, еще одна новая беда, в частности из-за того, что программисту приходится выполнять не свойственную ему задачу построения пользовательского интерфейса.

Применение «умных» указателей

Принципы использования «умных» указателей знакомы любому программисту на C++. Идея предельно проста: взамен того, чтобы пользоваться объектами некоторого класса, указателями на эти объекты или ссылками, определяется новый тип для которого переопределен селектор **->**, что позволяет использовать объекты такого типа в качестве ссылок на реальные объекты. На всякий случай, приведем следующий пример:

```
class A {
public:
   void method();
};

class APtr {
protected:
  A* a;
public:
  APtr();
  ~APtr();
  A* operator->();
};

inline APtr::APtr() : a(new A)
{ }

inline APtr::~APtr()
{
  delete a;
}

inline A* APtr::operator->()
{
  return a;
}
```

Теперь для объекта, определенного как

```
APtr: aptr;
```

можно использовать следующую форму доступа к члену **a**:

```
aptr->method();
```

Тонкости того, почему **operator->()** возвращает именно указатель **A*** (у которого есть свой селектор), а не, скажем, ссылку **A&** и все равно все компилируется таким образом, что выполнение доходит до метода **A::method()**, пропустим за ненадобностью — здесь мы не планируем рассказывать о том, как работает данный механизм и какие приемы применяются при его использовании.

Достоинства подобного подхода, в принципе, очевидны: возникает возможность контроля за доступом к объектам; малость тривиальных телодвижений и получается указатель, который сам считает количество используемых ссылок и при обнулении автоматически уничтожает свой объект, что позволяет не заботиться об этом самостоятельно... не важно? Почему же: самые трудно отлавливаемые ошибки — это ошибки в употреблении динамически выделенных объектов. Сплошь и рядом можно встретить попытку использования указателя на удаленный объект, двойное удаление объекта по одному и тому же адресу или неудаление объекта. При этом последняя ошибка, в принципе, самая невинная: программа, в которой не удаляются объекты (значит, теряется память, которая могла бы быть использована повторно) может вполне спокойно работать в течение некоторого периода (причем это время может спокойно колебаться от нескольких часов до нескольких дней), чего вполне хватает для решения некоторых задач. При этом заметить такую ошибку довольно просто: достаточно наблюдать динамику использования памяти программой; кроме того, имеются специальные средства для отслеживания подобных казусов, скажем, BoundsChecker.

Первая ошибка в данном списке тоже, в принципе, довольно элементарная: использование после удаления скорее всего приведет к тому, что операционная система скажет соответственное системное сообщение. Хуже становится тогда, когда подобного сообщения не возникает (т.е., данные достаточно правдоподобны или область памяти уже занята чем-либо другим), тогда программа может повести себя каким угодно образом.

Вторая ошибка может дать самое большое количество неприятностей. Все дело в том, что, хотя на первый взгляд она ничем особенным не отличается от первой, однако на практике вторичное удаление объекта приводит к тому, что менеджер кучи

удаляет что-то совсем немыслимое. Вообще, что значит «удаляет»? Это значит, что помечает память как пустую (готовую к использованию). Как правило, менеджер кучи, для того чтобы знать, сколько памяти удалить, в блок выделяемой памяти вставляет его размер. Так вот, если память уже была занята чем-то другим, то по «неверному» указателю находится неправильное значение размера блока, вследствие этого менеджер кучи удалит некоторый случайный размер используемой памяти. Это даст следующее: при следующих выделениях памяти (рано или поздно) менеджер кучи отдаст эту «неиспользуемую» память под другой запрос и... на одном клочке пространства будут ютиться два разных объекта. Крах программы произойдет почти обязательно, это лучшее что может произойти. Значительно хуже, если программа останется работать и будет выдавать правдоподобные результаты. Одна из самых оригинальных ошибок, с которой можно столкнуться и которая, скорее всего, будет вызвана именно повторным удалением одного и того же указателя, то, что программа, работающая несколько часов, рано или поздно «падет» в функции **malloc()**. Причем проработать она должна будет именно несколько часов, иначе эта ситуация не повторится.

Таким образом, автоматическое удаление при гарантированном неиспользовании указателя, это очевидный плюс. В принципе, можно позавидовать программистам на Java, у которых аналогичных проблем не возникает; зато, у них возникают другие проблемы.

Целесообразность использования «умных» указателей хорошо видно в примерах реального использования. Вот, к примеру, объявление «умного» указателя с подсчетом ссылок:

```
template<class T>
class MPtr
{
public:
  MPtr();
  MPtr(const MPtr<T>& p);
  ~MPtr();
  MPtr(T* p);

  T* operator->() const;
  operator T*() const;
```

```
    MPtr<T>& operator=(const MPtr<T>& p);
protected:
    struct RealPtr
    {
        T* pointer;
        unsigned int count;

        RealPtr(T* p = 0);
        ~RealPtr();
    };
    RealPtr* pointer;
private:
};
```

Особенно стоит оговорить здесь конструктор **MPtr::MPtr(T* p)**, который несколько выбивается из общей концепции. Все дело в том, что гарантировать отсутствие указателей на реальный объект может лишь создание такого объекта где-то внутри, это сделано в **MPtr::MPtr()**, где вызов **new** происходит самостоятельно. В итоге некоторая уверенность в том, что значение указателя никто нигде не сохранил без использования умного указателя, все-таки есть. Однако, очень нередко встречается такое, что у типа **T** может и не быть конструктора по умолчанию и объекту такого класса непременно при создании требуются какие-то аргументы для правильной инициализации. Совершенно правильным будет для подобного случая породить из **MPtr** новый класс, у которого будут такие же конструкторы, как и у требуемого класса. Оттого что подобный конструктор **MPtr::MPtr(T* p)** будет использоваться только лишь как **MPtr<T> ptr(new T(a,b,c))** и никак иначе, этот конструктор введен в шаблон.

Еще один спорный момент: присутствие оператора преобразования к **T***. Его наличие дает потенциальную возможность где-нибудь сохранить значение реального указателя.

Помимо **MPtr** можно использовать еще одну разновидность «умных» указателей, которая закономерно вытекает из описанной выше и отличается только лишь одной тонкостью:

```
template<class T>
class MCPtr
{
public:
```

```
  MCPtr(const MPtr<T>& p);
  MCPtr(const MCPtr<T>& p);
  ~MCPtr();

  const T* operator->() const;
  operator const T*() const;
  MCPtr<T>& operator=(const MPtr<T>& p)
  MCPtr<T>& operator=(const MCPtr<T>& p);
protected:
  MPtr<T> ptr;
private:
  MCPtr();
};
```

Во-первых, это надстройка (адаптер) над обычным указателем. А во-вторых, его главное отличие, это то, что **operator->** возвращает константный указатель, а не обычный. Это очень просто и, на самом деле, очень полезно: все дело в том, что это дает использовать объект в двух контекстах — там, где его можно изменять (скажем, внутри другого объекта, где он был создан) и там, где можно пользоваться лишь константным интерфейсом (т.е., где изменять нельзя; к примеру, снаружи объекта-фабрики). Это разумно вытекает из простых константных указателей. Для того, чтобы пользоваться **MCPtr** требуется единственное (хотя и достаточно строгое) условие: во всех классах должна быть корректно расставлена константность методов. Вообще, это — признак профессионального программиста: использование модификатора **const** при описании методов.

Как правило используют «умные» указатели в том, что называется фабриками объектов (или, в частности, производящими функциями): т.е., для того, чтобы вернуть объект, удовлетворяющий какому-то интерфейсу. При использовании подобного рода указателей клиентской части становится очень удобно — опускаются все проблемы, связанные с тем, когда можно удалить объект, а когда нельзя (скажем, при совместном использовании одного и того же объекта разными клиентами — клиенты не обязаны знать о существовании друг друга).

Помимо всего прочего, переопределение селектора позволяет простым образом вставить синхронизацию при создании многопоточных приложений. Вообще, подобные

«обертки» чрезвычайно полезны, им можно найти массу применений.

Несомненно, использовать «умные» указатели необходимо с осторожностью. Все дело в том, что, как у всякой простой идеи, у нее есть один очень большой недостаток: несложно придумать пример, в котором два объекта ссылаются друг на друга через «умные» указатели и... никогда не будут удалены. Почему? Потому что счетчики ссылок у них всегда будут как минимум 1, при том, что снаружи на них никто не ссылается. Есть рекомендации по поводу того, как распознавать такие ситуации во время выполнения программы, но они очень громоздки и, поэтому не годятся к применению. Ведь что прельщает в «умных» указателях? Простота. Фактически, ничего лишнего, а сколько можно при желании извлечь пользы из их применения.

Посему надо тщательно следить еще на стадии проектирования за тем, чтобы подобных цепочек не могло бы возникнуть в принципе. Потому как если такая возможность будет, то в конце концов она проявит себя.

«Умные» указатели активно используются в отображении COM-объектов и CORBA-объектов на C++: они позволяют прозрачно для программиста организовать работу с объектами, которые реально написаны на другом языке программирования и выполняются на другой стороне земного шара.

Техника подсчета ссылок в явном виде (через вызов методов интерфейса AddRef() и Release()) используется в технологии COM.

Еще стоит сказать про эффективность использования «умных» указателей. Возможно, это кого-то удивит, но затраты на их использование при выполнении программы минимальны. Почему? Потому что используются шаблоны, а все методы-члены классов (и, в особенности селектор) конечно же объявлены как **inline**. Подсчет ссылок не сказывается на обращении к объекту, только на копировании указателей, а это не такая частая операция. Ясно, что использование шаблонов усложняет работу компилятора, но это не так важно.

Рассуждения на тему «Умных» указателей

При изучении C++, не раз можно встретиться с «умными» указателями. Они встречаются везде и все время в разных вариантах. Без стандартизации.

Вообще, мысль об упрощении себе жизни вертится в головах программистов всегда: «Лень — двигатель прогресса». Поэтому и были придуманы не просто указатели, а такие из них, которые брали часть умственного напряга на себя, тем самым, делая вид, что они нужны.

Итак, что такое **SmartPointer**-ы? По сути это такие классы, которые умеют чуть больше... — а в общем, смотрим пример:

```
class A
{
  private:
    int count;
  public:
    A(){count = 0;}
    void addref(){count++;}
    void release(){if(--count == 0) delete this;}
  protected:
    ~A();
  public:
    void do_something(){cout << "Hello";}
};
```

Сначала придумали внутри объекта считать ссылки на него из других объектов при помощи «механизма подсчета ссылок». Суть здесь в том, что когда вы сохраняете ссылку на объект, то должны вызвать для него **addref**, а когда избавляетесь от объекта, то вызвать **release**. Сложно? Совсем нет — это дело привычки. Таким образом, объект умеет сам себя удалять. Здорово? Так оно и есть.

Кстати такой объект может существовать только в куче, поскольку деструктор в «защищенной» зоне и по той же причине нельзя самому сделать «**delete a**» обойдя **release**.

Теперь переходим к собственно самим «умным» указателям и опять пример:

```
class PA
{
```

```
  private:
    A* pa;
  public:
    PA(A* p){pa = p; p->addref();}
    ~PA(){pa->release();}
    A* operator ->(){return pa;}
};
```

Что мы видим? Есть класс **PA**, который умеет принимать нормальный указатель на объект класса **A** и выдавать его же по обращению к селектору членов класса. «Ну и что? — скажете вы, — Как это может помочь?». За помощью обратимся к двум примерам, которые иллюстрирует эффективность использования класса **PA**:

```
...
{
  A* pa = new A();
  pa->addref();
  pa->do_something();
  pa->release();
}


...
{
  PA pa = new A();
  pa->do_something();
}
```

Посмотрим внимательнее на эти два отрывка... Что видим? Видим экономию двух строчек кода. Здесь вы наверное скажете: «И что, ради этого мы столько старались?». Но это не так, потому что с введением класса **PA** мы переложили на него все неприятности со своевременными вызовами **addref** и **release** для класса **A**. Вот это уже что-то стоит!

Дальше больше, можно добавить всякие нужные штучки, типа оператора присваивания, еще одного селектора (или как его некоторые называют «разименователь» указателя) и так далее. Таким образом получится удобная вещь (конечно, если все это дело завернуть в шаблон.

Теперь немного сменим направление рассуждений. Оказывается существуют такие вещи, как «Мудрые указатели»,

«Ведущие указатели», «Гениальные указатели», «Грани», «Кристаллы» — в общем, хватает всякого добра. Правда, в практичности этих поделок можно усомниться, несмотря на их «изящество». То есть, конечно, они полезны, но не являются, своего рода, панацеей (кроме «ведущих» указателей).

В общем особую роль играют только «ведущие» и «умные» указатели.

Начнем с такого класса как **Countable**, который будет отвечать за подсчет чего либо.

Итак он выглядит примерно так (в дальнейшем будем опускать реализации многих функций из-за их тривиальности, оставляя, тем самым, только их объявления:

```
class Countable
{
  private:
    int count;
  public:
    int increment ();
    int decrement ();
};
```

Здесь особо нечего говорить, кроме того, что, как всегда, этот класс можно сделать более «удобным», добавив такие вещи, как поддержку режима многопоточности и т.д.

Следующий простой класс прямо вытекает из многопоточности и осуществляет поддержку этого режима для своих детей:

```
class Lockable
{
  public:
    void lock();
    void unlock();
    bool islocked();
};
```

Этот класс не вносит никакой новизны, но стандартизует поддержку многопоточности при использовании, например, различных платформ.

Теперь займемся собственно указателями:

```
class AutoDestroyable : public Countable
{
  public:
    virtual int addref ();
    virtual int release ();
  protected:
    virtual ~AutoDestroyable();
    ...
};
```

Из кода видно, что этот класс занимается подсчетом ссылок и «убивает» себя если «пришло время».

А сейчас процитируем код «умного» указателя, для того чтобы синхронизовать наши с вами понимание этого чуда.

```
template <class T>
class SP
{
  private:
    T* m_pObject;
  public:
    SP ( T* pObject){ m_pObject = pObject; }
    ~SP () { if( m_pObject ) m_pObject->release (); }
    T* operator -> ();
    T& operator * ();
    operator T* ();
    ...
};
```

Это уже шаблон, он зависит от объекта класса, к которому применяется. Задача «умного» указателя была рассмотрена выше и итог при сравнении с ситуацией без его использования положителен только тем, что для объекта, создаваемого в куче, не надо вызывать оператор **delete** — он сам вызовется, когда это понадобится.

Теперь остановимся на минутку и подумаем, когда мы можем использовать этот тип указателей, а когда нет. Главное требование со стороны **SP**, это умение основного объекта вести подсчет ссылок на себя и уничтожать себя в тот момент, когда он не становится нужен. Это серьезное ограничение, поскольку не во все используемые классы вы сможете добавить эти

возможности. Вот несколько причин, по которым вы не хотели бы этого (или не можете):

- Вы используете закрытую библиотеку (уже скомпилированную) и физически не можете добавить кусок кода в нее.
- Вы используете открытую библиотеку (с исходными кодами), но не хотите изменять ее как-либо, потому что все время меняете ее на более свежую (кто-то ее делает и продвигает за вас).
- И, наконец, вы используете написанный вами класс, но не хотите по каким-либо причинам вставлять поддержку механизма подсчета ссылок.

Итак, причин много или по крайней мере достаточно для того, чтобы задуматься над более универсальным исполнением **SP**. Посмотрим на схематичный код «ведущих» указателей и «дескрипторов»:

```
template <class T>
clacc MP : public AutoDestroyable
{
  private:
    T* m_pObj;
  public:
    MP(T* p);
    T* operator ->();
  protected:
    operator T*();
};

template <class T>
class H
{
  private:
    MP<T>* m_pMP;
  public:
    H(T*);
    MP& operator T->();
    bool operator ==(H<T>&);
    H operator =(H<T>&);
};
```

Что мы видим на этот раз? А вот что. **MP** — это «ведущий» указатель, т.е. такой класс, который держит в себе объект основного класса и не дает его наружу. Появляется только вместе с ним и умирает аналогично. Его главная цель, это реализовывать механизм подсчета ссылок.

В свою очередь **H** — это класс очень похожий на **SP**, но общается не с объектом основного класса, а с соответствующим ему **MP**.

Результат этого шаманства очевиден — мы можем использовать механизм «умных» указателей для классов не добавляя лишнего кода в их реализацию.

И это действительно так, ведь широта использования такого подхода резко увеличивается и уже попахивает панацеей.

Что же можно сказать о многопоточности и множественном наследовании: при использовании классов **MP** и **H** — нет поддержки этих двух вещей. И если с первой все понятно (нужно наследовать MP от Lockable), то со вторым сложнее.

Итак, посвятим немного времени описанию еще одного типа указателей, которые призваны «решить» проблему множественного наследования (а точнее полиморфизма).

Рассмотрим классы **PP** и **TP**:

```
class PP : public AutoDestroyable
{};

template<class T>
class TP
{
  protected:
    T* m_pObject;
    PP* m_pCounter;

  public:
    TP ();
    TP ( T* pObject );
    TP<T>& operator = ( TP<T>& tp );
    T* operator -> ();
    T& operator * ();
    operator T* ();
```

```
    bool operator == ( TP<T>& tp );
};
```

Класс **PP** является «фиктивным ребенком» **AutoDestroyable** и вы не забивайте себе этим голову. А вот класс **TP** можно посмотреть и попристальнее.

Схема связей в этом случае выглядит уже не **H->MP->Obj**, а **PP<-TP->Obj**, т.е. Счетчик ссылок (а в данном случае, это **PP**) никак не связан с основным объектом или каким-либо другим и занимается только своим делом — ссылками. Таким образом, на класс **TP** ложится двойная обязанность: выглядеть как обычный указатель и отслеживать вспомогательные моменты, которые связаны со ссылками на объект.

Как же нам теперь использовать полиморфизм? Ведь мы хотели сделать что-то вроде:

```
class A : public B
...
TP<A> a;
TP<B> b;
a = new B;
b = (B*)a;
...
```

Для этого реализуем следующую функцию (и внесем небольшие изменения в класс **TP** для ее поддержки):

```
template <class T, class TT>
TP<T> smart_cast ( TP<TT>& tpin );
```

Итак, теперь можно написать что-то вроде (и даже будет работать):

```
class A : public B
...
TP<A> a;
TP<B> b;
a = new B;
b = smart_cast<B, A>(a);

// или если вы используете Visual C++, то даже
b = smart_cast<B>(a);
...
```

Вам ничего не напоминает? Ага, схема та же, что и при использовании **static_cast** и **dynamic_cast**. Так как схожесть очень убедительна, можно заключить, что такое решение проблемы более чем изящно.

Виртуальные деструкторы

В практически любой мало-мальски толковой книге по C++ рассказывается, зачем нужны виртуальные деструкторы и почему их надо использовать. При всем при том, как показывает практика, ошибка, связанная с отсутствием виртуальных деструкторов, повсеместно распространена.

Итак, рассмотрим небольшой пример:

```
class A
{
public:
  virtual void f() = 0;
  ~A();
};
class B : public A
{
public:
  virtual void f();
  ~B();
};
```

Вызов компилятора **gcc** строкой:

```
g++ -c -Wall test.cpp
```

даст следующий результат:

```
test.cpp:6: warning: `class A' has virtual functions but
non-virtual destructor
test.cpp:13: warning: `class B' has virtual functions but
non-virtual destructor
```

Это всего лишь предупреждения, компиляция прошла вполне успешно. Однако, почему же **gcc** выдает подобные предупреждения?

Все дело в том, что виртуальные функции используются в C++ для обеспечения полиморфизма — т.е., клиентская функция вида:

```
void call_f(A* a)
{
  a->f();
}
```

никогда не «знает» о том, что конкретно сделает вызов метода **f()** — это зависит от того, какой в действительности объект представлен указателем **a**. Точно так же сохраняются указатели на объекты:

```
std::vector<A*> a_collection;
a_collection.push_back(new B());
```

В результате такого кода теряется информация о том, чем конкретно является каждый из элементов **a_collection** (имеется в виду, без использования RTTI). В данном случае это грозит тем, что при удалении объектов:

```
for(std::vector<A*>::iterator i = ... )
  delete *i;
```

все объекты, содержащиеся в **a_collection**, будут удалены так, как будто это — объекты класса **A**.

В этом можно убедиться, если соответствующим образом определить деструкторы классов **A** и **B**:

```
inline A::~A()
{
  puts("A::~A()");
}

inline B::~B()
{
  puts("B::~B()");
}
```

Тогда выполнение следующего кода:

```
A* ptr = new B();
delete ptr;
```

приведет к следующему результату:

```
A::~A()
```

Если же в определении класса **A** деструктор был бы сделан виртуальным (**virtual ~A();**), то результат был бы другим:

```
B::~B()
A::~A()
```

В принципе, все сказано. Но, несмотря на это, очень многие программисты все равно не создают виртуальных деструкторов. Одно из распространенных заблуждений — виртуальный деструктор необходим только в том случае, когда на деструктор порожденных классов возлагаются какие-то нестандартные функции; если же функционально деструктор порожденного класса ничем не отличается от деструктора предка, то делать его виртуальным совершенно необязательно. Это неправда, потому что даже если деструктор никаких специальных действий не выполняет, он все равно должен быть виртуальным, иначе не будут вызваны деструкторы для объектов-членов класса, которые появились по отношению к предку. То есть:

```
#include <stdio.h>

class A
{
public:
  A(const char* n);
  ~A();
protected:
  const char* name;
};

inline A::A(const char* n) : name(n)
{ }
inline A::~A()
{
  printf("A::~A() for %s.\n", name);
}

class B
{
public:
  virtual void f();
  B();
  ~B();
protected:
  A a1;
};
```

```
inline B::~B()
{ }

inline B::B() : a1("a1")
{ }
void B::f() { }
class C : public B
{
public:
  C();
protected:
  A a2;
};

inline C::C() : a2("a2")
{ }

int main()
{
 B* ptr = new C();
 delete ptr;
 return 0;
}
```

Компиляция данного примера проходит без ошибок (но с предупреждениями), вывод программы следующий:

```
A::~A() for a1
```

Немного не то, что ожидалось? Тогда поставим перед названием деструктора класса **B** слово **virtual**. Результат изменится:

```
A::~A() for a2
A::~A() for a1
```

Сейчас вывод программы несколько более соответствует действительности.

Запись структур данных в двоичные файлы

Чтение и запись данных, вообще говоря, одна из самых часто встречающихся операций. Сложно себе представить программу, которая бы абсолютно не нуждалась бы в том, чтобы отобразить где-нибудь информацию, сохранить промежуточные

данные или, наоборот, восстановить состояние прошлой сессии работы с программой.

Собственно, все эти операции достаточно просто выполняются — в стандартной библиотеке любого языка программирования обязательно найдутся средства для обеспечения ввода и вывода, работы с внешними файлами. Но и тут находятся некоторые сложности, о которых, обычно, не задумываются.

Итак, как все это выглядит обычно? Имеется некоторая структура данных:

```
struct data_item
{
  type_1 field_1;
  type_2 field_2;
  // ...
  type_n field_n;
};
data_item i1;
```

Каким образом, например, сохранить информацию из **i1** так, чтобы программа во время своего повторного запуска, смогла восстановить ее? Наиболее частое решение следующее:

```
FILE* f = fopen("file", "wb");
fwrite((char*)&i1, sizeof(i1), 1, f);
fclose(f);
```

assert расставляется по вкусу, проверка инвариантов в данном примере не является сутью. Тем не менее, несмотря на частоту использования, этот вариант решения проблемы не верен.

Нет, он будет компилироваться и, даже будет работать. Мало того, будет работать и соответствующий код для чтения структуры:

```
FILE* f = fopen("file", "rb");
fread((char*)&i1, sizeof(i1), 1, f);
fclose(f);
```

Что же тут неправильного? Ну что же, для этого придется немного пофилософствовать. Как бы много не говорили о том, что Си — это почти то же самое, что и ассемблер, не надо забывать, что он является все-таки языком высокого уровня. Следовательно, в принципе, программа написанная на Си (или C++) может (теоретически) компилироваться на разных

компиляторах и разных платформах. К чему это? К тому, что данные, которые сохранены подобным образом, в принципе не переносимы.

Стоит вспомнить о том, что для структур неизвестно их физическое представление. То есть, для конкретного компилятора оно, быть может, и известно (для этого достаточно посмотреть работу программы «вооруженным взглядом», т.е. отладчиком), но о том, как будут расположены в памяти поля структуры на какой-нибудь оригинальной машине, неизвестно. Компилятор со спокойной душой может перетасовать поля (это, в принципе, возможно) или выровнять положение полей по размеру машинного слова (встречается сплошь и рядом). Для чего? Для увеличения скорости доступа к полям. Понятно, что если поле начинается с адреса, не кратного машинному слову, то прочитать его содержимое не так быстро, как в ином случае. Таким образом, сохранив данные из памяти в бинарный файл напрямую мы получаем дамп памяти конкретной архитектуры (не говоря о том, что **sizeof** совершенно не обязан возвращать количество байт).

Плохо это тем, что при переносе данных на другую машину при попытке прочитать их той же программой (или программой, использующую те же структуры) вполне можно ожидать несколько некорректных результатов. Это связано с тем, что структуры могут быть представлены по другому в памяти (другое выравнивание), различается порядок следования байтов в слове и т.п. Как этого избежать?

Обычный «костыль», который применяется, например, при проблемах с выравниванием, заключается в том, что компилятору явно указывается как надо расставлять поля в структурах. В принципе, любой компилятор дает возможность управлять выравниванием. Но выставить одно значение для всего проекта при помощи ключей компилятора (обычно это значение равно 1, потому что при этом в сохраненном файле не будет пустых мест) нехорошо, потому что это может снизить скорость выполнения программы. Есть еще один способ указания компилятору размера выравнивания, он заключается в использовании директивы препроцессора **#pragma**. Это не оговорено стандартом, но обычно есть директива **#pragma pack**, позволяющая сменить выравнивание для определенного отрезка исходного текста. Выглядит это обычно примерно так:

```
#pragma pack(1)
struct { /* ... */ };
#pragma pack(4)
```

Последняя директива **#pragma pack(4)** служит для того, чтобы вернуться к более раннему значению выравнивания. В принципе, конечно же при написании исходного текста никогда доподлинно заранее неизвестно, какое же было значение выравнивания до его смены, поэтому в некоторых компиляторах под Win32 есть возможность использования стека значений (пошло это из MS Visual C++):

```
#pragma pack(push, 1)
struct { /* ... */ };
#pragma pack(pop)
```

В примере выше сначала сохраняется текущее значение выравнивания, затем оно заменяется 1, затем восстанавливается ранее сохраненное значение. При этом, подобный синтаксис поддерживает даже **gcc** для win32 (еще стоит заметить, что, вроде, он же под Unix использовать такую запись **#pragma pack** не дает). Есть альтернативная форма **#pragma pack()**, поддерживаемая многими компиляторами (включая **msvc** и **gcc**), которая устанавливает значение выравнивания по умолчанию.

И, тем не менее, это не хорошо. Опять же, это дает очень интересные ошибки. Представим себе следующую организацию исходного текста. Сначала заголовочный файл **inc.h**:

```
#ifndef __inc_h__
#define __inc_h__

class Object
{
  // ...
};

#endif // __inc_h__
```

Представьте себе, что существуют три файла **file1.cpp**, **file2.cpp** и **file2.h**, которые этот хидер используют. Допустим, что в **file2.h** находится функция **foo**, которая (например) записывает **Object** в файл:

```
// file1.cpp
#include "inc.h"
```

```
#include "file2.h"

int main()
{
  Object* obj = new Object();
  foo(obj, "file");

  delete obj;

  return 0;
}

// file2.h
#ifndef __file2_h__
#define __file2_h__

#pragma pack(1)

#include "inc.h"

void foo(const Object* obj, const char* fname);

#pragma pack(4)

#endif // __file2_h__

// file2.cpp
#include "file2.h"

void foo(const Object* obj, const char* fname)
{
  // ...
}
```

Это все скомпилируется, но работать не будет. Почему? Потому что в двух разных единицах компиляции (**file1.cpp** и **file2.cpp**) используется разное выравнивание для одних и тех же структур данных (в данном случае, для объектов класса **Object**). Это даст то, что объект переданный по указателю в функцию **foo()** из функции **main()** будет разным (и, конечно же, совсем неправдоподобным). Понятно, что это явный пример «плохой» организации исходных текстов — использование директив

компилятора при включении заголовочных файлов, но, поверьте, он существует.

Отладка программы, содержащую подобную ошибку, оказывается проверкой на устойчивость психики. Потому что выглядит это примерно так: следим за объектом, за его полями, все выглядит просто замечательно и вдруг, после того как управление передается какой-то функции, все, что содержится в объекте, принимает «бредовые» формы, какие-то неизвестно откуда взявшиеся цифры...

На самом деле **#pragma pack** не является панацеей. Мало того, использование этой директивы практически всегда неправомерно. Можно даже сказать, что эта директива в принципе редко когда нужна (во всяком случае, при прикладном программировании).

Правильным же подходом является сначала записать все поля структуры в нужном порядке в некоторый буфер и скидывать в файл уже содержимое буфера. Это очень просто и очень эффективно, потому что все операции чтения/записи можно собрать в подпрограммы и менять их при необходимости таким образом, чтобы обеспечить нормальную работу с внешними файлами. Проиллюстрируем этот подход:

```
template<class T>
inline size_t get_size(const T& obj)
{
  return sizeof(obj);
}
```

Эта функция возвращает размер, необходимый для записи объекта. Зачем она понадобилась? Во-первых, возможен вариант, что **sizeof** возвращает размер не в байтах, а в каких-то собственных единицах. Во-вторых, и это значительно более необходимо, объекты, для которых вычисляется размер, могут быть не настолько простыми, как **int**. Например:

```
template<>
inline size_t get_size<std::string>(const std::string& s)
{
  return s.length() + 1;
}
```

Надеемся, понятно, почему выше нельзя было использовать **sizeof**.

Аналогичным образом определяются функции, сохраняющие в буфер данные и извлекающие из буфера информацию:

```
typedef unsigned char byte_t;
template<class T>
inline size_t save(const T& i, byte_t* buf)
{
  *((T*)buf) = i;
  return get_size(i);
}

template<class T>
inline size_t restore(T& i, const byte_t* buf)
{
  i = *((T*)buf);
  return get_size(i);
}
```

Понятно, что это работает только для простых типов (**int** или **float**), уж очень много чего наворочено: явное приведение указателя к другому типу, оператор присваивания... конечно же, очень нехорошо, что такой **save()** доступен для всех объектов. Понятно, что очень просто от него избавиться, убрав шаблонность функции и реализовав аналогичный **save()** для каждого из простых типов данных. Тем не менее, это всего-лишь примеры использования:

```
template<>
inline size_t save<MyObject>(const MyObject& s, byte_t* buf)
{
  // ...
}
```

Можно сделать и по другому. Например, ввести методы **save()** и **restore()** в каждый из сохраняемых классов, но это не столь важно для принципа этой схемы. Поверьте, это достаточно просто использовать, надо только попробовать. Мало того, здесь можно вставить в **save<long>()** вызов **htonl()** и в **restore<long>()** вызов **ntohl()**, после чего сразу же упрощается перенос двоичных файлов на платформы с другим порядком байтов в слове... в общем, преимуществ — море. Перечислять все из них не стоит, но как после этого лучше выглядит исходный текст, а как приятно вносить изменения...

Оператор безусловного перехода goto

Так уж сложилось, что именно присутствие или отсутствие этого оператора в языке программирования всегда вызывает жаркие дебаты среди сторонников «хорошего стиля» программирования. При этом, и те, кто «за», и те, кто «против» всегда считают признаком «хорошего тона» именно использование **goto** или, наоборот, его неиспользование. Не вставая на сторону ни одной из этих «школ», просто покажем, что действительно есть места, где использование **goto** выглядит вполне логично.

Но сначала о грустном. Обычно в вину **goto** ставится то, что его присутствие в языке программирования позволяет делать примерно такие вещи:

```
int i, j;
for(i = 0; i < 10; i++)
{
  // ...

  if(condition1)
    {
      j = 4;
      goto label1;
    }

  // ...

  for(j = 0; j < 10; j++)
  {
    // ...
  label1:
    // ...
    if(condition2)
      {
        i = 6;
        goto label2;
      }
  }

  // ...
```

```
label2:
  // ...
}
```

Прямо скажем, что такое использование **goto** несколько раздражает, потому что понять при этом, как работает программа при ее чтении будет очень сложно. А для человека, который не является ее автором, так и вообще невозможно. Понятно, что вполне вероятны случаи, когда такого подхода требует какая-нибудь очень серьезная оптимизация работы программы, но делать что-то подобное программист в здравом уме не должен. На самом деле, раз уж мы привели подобный пример, в нем есть еще один замечательный нюанс — изменение значения переменной цикла внутри цикла. Смеем вас заверить, что такое поведение вполне допустимо внутри **do** или **while**; но когда используется **for —** такого надо избегать, потому что отличительная черта **for** как раз и есть жестко определенное местоположение инициализации, проверки условия и инкремента (т.е., изменения переменной цикла). Поэтому читатель исходного текста, увидев «полный» **for** (т.е. такой, в котором заполнены все эти три места) может и не заметить изменения переменной где-то внутри цикла. Хотя для циклов с небольшим телом это, наверное, все-таки допустимо — такая практика обычно применяется при обработке строк (когда надо, например, считать какой-то символ, который идет за «спецсимволом», как «\\» в строках на Си; вместо того, чтобы вводить дополнительный флаг, значительно проще, увидев «\», сразу же сдвинуться на одну позицию и посмотреть, что находится там). В общем, всегда надо руководствоваться здравым смыслом и читабельностью программы.

Если здравый смысл по каким-то причинам становится в противовес читабельности программы, то это место надо обнести красными флагами, чтобы читатель сразу видел подстерегающие его опасности.

Тем не менее, вернемся к **goto**. Несмотря на то, что такое расположение операторов безусловного перехода несколько нелогично (все-таки, вход внутрь тела цикла это, конечно же, неправильно) — это встречается.

Итак, противники использования **goto** в конечном итоге приходят к подобным примерам и говорят о том, что раз такое его

использование возможно, то лучше чтобы его совсем не было. При этом, конечно же, никто обычно не спорит против применения, например, **break**, потому что его действие жестко ограничено. Хочется сказать, что подобную ситуацию тоже можно довести до абсурда, потому что имеются программы, в которых введен цикл только для того, чтобы внутри его тела использовать **break** для выхода из него (т.е., цикл делал только одну итерацию, просто в зависимости от исходного состояния заканчивался в разных местах). И что помешало автору использовать **goto** (раз уж хотелось), кроме догматических соображений, не понятно.

Собственно, мы как раз подошли к тому, что обычно называется «разумным» применением этого оператора. Вот пример:

```
switch(key1)
{
case q1 :
     switch(key2)
     {
        case q2 : break;
     }
     break;
}
```

Все упрощено до предела, но, в принципе, намек понятен. Есть ситуации, когда нужно что-то в духе **break**, но на несколько окружающих циклов или операторов **switch**, а **break** завершает только один. Понятно, что в этом примере читабельность, наверное, не нарушена (в смысле, использовался бы вместо внутреннего **break goto** или нет), единственное, что в таком случае будет выполнено два оператора перехода вместо одного (**break** это, все-таки, разновидность **goto**).

Значительно более показателен другой пример:

```
bool end_needed = false;
for( ... )
{
    for( ... )
    {
        if(cond1) { end_needed = true; break; }
    }
```

```
    if(end_needed) break;
}
```

Т.е., вместо того, чтобы использовать **goto** и выйти из обоих циклов сразу, пришлось завести еще одну переменную и еще одну проверку условия. Тут хочется сказать, что **goto** в такой ситуации выглядит много лучше — сразу видно, что происходит; а то в этом случае придется пройти по всем условиями и посмотреть, куда они выведут. Надо сказать (раз уж мы начали приводить примеры из жизни), что не раз можно видеть эту ситуацию, доведенную до крайности — четыре вложенных цикла (ну что поделать) и позарез надо инициировать выход из самого внутреннего. И что? Три лишних проверки... Кроме того, введение еще одной переменной, конечно же, дает возможность еще раз где-нибудь допустить ошибку, например, в ее инициализации. Опять же, читателю исходного текста придется постоянно лазить по тексту и смотреть, зачем была нужна эта переменная... в общем: не плодите сущностей без надобности. Это только запутает.

Другой пример разумного использования **goto** следующий:

```
int foo()
{
  int res;

  // ...
  if(...)
  {
     res = 10;
     goto finish;
  }
  // ...

finish:
  return res;
}
```

Понятно, что без **goto** это выглядело бы как **return 10** внутри **if**. Итак, в чем преимущества такого подхода. Ну, сразу же надо вспомнить про концептуальность — у функции становится только один «выход», вместо нескольких (быстро вспоминаем про IDEF). Правда, концептуальность — это вещь такая... Неиспользование **goto** тоже в своем роде концептуальность, так что это не показатель (нельзя противопоставлять догму догме, это

просто глупо). Тем не менее, выгоды у такого подхода есть. Во-первых, вполне вероятно, что перед возвратом из функции придется сделать какие-то телодвижения (закрыть открытый файл, например). При этом, вполне вероятно, что когда эта функция писалась, этого и не требовалось — просто потом пришлось дополнить. И что? Если операторов **return** много, то перед каждым из них появится одинаковый кусочек кода. Как это делается? Правильно, методом «cut&paste». А если потом придется поменять? Тоже верно, «search&replace». Объяснять, почему это неудобно не будем — это надо принять как данность.

Во-вторых, обработка ошибок, которая также требует немедленного выхода с возвратом используемых ресурсов. В принципе, в C++ для этого есть механизм исключительных ситуаций, но когда он отсутствует (просто выключен для повышения производительности), это будет работать не хуже. А может и лучше по причине более высокой скорости.

В-третьих, упрощается процесс отладки. Всегда можно проверить что возвращает функция, поставить точку останова (хотя сейчас некоторые отладчики дают возможность найти все **return** и поставить на них точки останова), выставить дополнительный **assert** или еще что-нибудь в этом духе. В общем, удобно.

Еще **goto** очень успешно применятся при автоматическом создании кода — читателя исходного текста там не будет, он будет изучать то, по чему исходный текст был создан, поэтому можно (и нужно) допускать различные вольности.

В заключение скажем, что при правильном использовании оператор **goto** очень полезен. Надо только соблюдать здравый смысл, но это общая рекомендация к программированию на C/C++ (да и вообще, на любом языке программирования), поэтому непонятно почему **goto** надо исключать.

Виртуальный конструктор

Предупреждение: то, что описано — не совсем уж обычные объекты. Возможно только динамическое их создание и только в отведённой уже памяти. Ни о каком статическом или автоматическом их создании не может быть и речи. Это не цель и не побочный эффект, это расплата за иные удобства.

Краткое описание конкретной ситуации, где всё это и происходило. В некотором исследовательском центре есть биохимическая лаборатория. Есть в ней куча соответствующих анализаторов. Железки они умные, работают самостоятельно, лишь бы сунули кассету с кучей материалов и заданиями. Всякий анализатор обрабатывает материалы только определённой группы. Со всех них результаты текут по одному шлангу в центр всяческих обработок и складирования. Масса частностей, но нам они неинтересны. Суть — всякий результат есть результат биохимического анализа. Текущий потоком байт с соответствующими заголовками и всякими телами. Конкретный тип реально выяснить из первых шестнадцати байт. Максимальный размер — есть. Но он лишь максимальный, а не единственно возможный.

Не вдаваясь в подробности принятого решения (это вынудит вдаваться в подробности задания), возникла конкретная задача при реализации — уже во время исполнения конструктора объекта конкретный тип результата анализа неизвестен. Неизвестен (соответственно) и его размер. Известен лишь размер пула для хранения некоторого количества этих объектов. Две проблемы — сконструировать объект конкретного типа в конструкторе объекта другого (обобщающего) типа и положить его на это же самое место в памяти. При этом память должно использовать эффективно, всячески минимизируя (главная проблема) фрагментацию пула, ибо предсказать время, в течение которого результат будет оставаться нужным (в ожидании, в частности, своих попутчиков от других анализаторов), невозможно. Это — не очередь (иначе всё было бы значительно проще).

Решение: от классической идиомы envelope/letter (которая сама по себе основа кучи идиом) к «виртуальному» конструктору с особым (либо входящим в состав, либо находящимся в дружеских отношениях) менеджером памяти. Излагается на смеси C++ и недомолвок (некритичных) в виде «. . .»:

```
class BCAR {     // Bio-Chemical Analysis Result

friend class BCAR_MemMgr;

protected:
    BCAR() { /* Должно быть пусто!!! */ }
```

```
public:
     BCAR( const unsigned char * );
     void *operator new( size_t );
     void operator delete( void * );
     virtual int size() { return 0; }
     . . .

private:
     struct {
          // трали-вали
     } header;
     . . .
};
```

Это был базовый класс для всех прочих конкретных результатов анализов. У него есть свой **new**. Но для реализации идеи используется не дефолтовый **new** из C++ **rtl**, а используется следующее:

```
inline void *operator new( size_t, BCAR *p ) {
     return p;
}
```

Именно за счёт его мы получим **in place** замену объекта одного класса (базового) объектом другого (производного). Раньше было проще — **this** допускал присваивание.

Теперь — менеджер памяти.

```
class BCAR_MemMgr {

friend BCAR;

public:
     BCAR_MemMgr();
     void alloc( int );
     void free( BCAR *, int );
     BCAR *largest();

private:
     . . .
};
```

Это примерный его вид. Он создаётся в единственном экземпляре:

```
static BCAR_MemMgr MemoryManager;
```

и занимается обслугой пула памяти под все объекты. В открытом интерфейсе у него всего три функции, назначение **alloc/free** любому понятно (хотя **alloc** в действительности ничего не аллоцирует, а делает «обрезание» того, что даёт **largest** и соответствующим образом правит списки менеджера), а **largest** возвращает указатель на самый большой свободный блок. В сущности, она и есть **BCAR::new**, которая выглядит так:

```
void *BCAR::operator new( size_t ) {
    return MemoryManager.largest();
}
```

Зачем самый большой? А затем, что при создании объекта его точный тип ещё неизвестен (ибо создаваться будет через **new BCAR**), поэтому берём по максимуму, а потом **alloc** всё подправит.

Теперь собственно классы для конкретных результатов. Все они выглядят примерно одинаково:

```
class Phlegm: public BCAR {

friend BCAR;

private:
    int size() { retrurn sizeof( Phlegm ); }
    struct PhlegmAnalysisBody {
        // тут всякие его поля
    };
    PhlegmAnalysisBody body;
    Phlegm( const unsigned char *data ): BCAR() {
        MemoryManager.alloc( size() );
        ::memcpy( &body, data + sizeof( header ),
        sizeof( body ) );
    }
    . . .
};
```

Где тут расположен «виртуальный» конструктор. А вот он:

```
BCAR::BCAR( const unsigned char *dataStream ) {
    ::memcpy( &header, dataStream, sizeof( header ) );
```

```
        if( CRC_OK( dataStream ) ) {
             // определяем тип конкретного результата
             // и строим соответствующий объект прямо на месте
себя
             switch( AnalysisOf( dataStream ) ) {
case PHLEGM:
     ::new( this ) Phlegm( dataStream );
     break;
case BLOOD:
     ::new( this ) Blood( dataStream );
     break;
case ...:
     . . .
          }
     . . .
}
```

Теперь, чтобы не вынуждать вас носиться по всему материалу в целях построения целостной картины, объясним происходящее по шагам.

Менеджер памяти создан, инициализирован. Пул памяти существует (хотя бы от обычного **malloc**, а хоть и с потолка — ваше дело). Есть некоторый поток байт (пусть он зовётся **stream**), в котором то, с чем мы и боремся. Объект создаётся следующим образом:

```
BCAR *analysis = new BCAR( stream );
```

Обратите внимание — мы создаём объект класса **BCAR**. В первую очередь вызывается **BCAR::new**, который в действительности завуалированный **MemoryManager.largest()**. Мы имеем адрес в свободной памяти, где и создаётся объект **BCAR** и запускается его конструктор **BCAR::BCAR(const unsigned char *)**. В конструкторе по информации из заголовка (полученного из потока **stream**) выясняется точный тип анализа и через глобальный **new** (который не делает ничего) создаётся на месте объекта **BCAR** объект уточнённого типа. Начинает исполняться его конструктор, который в свою очередь вызывает конструктор **BCAR::BCAR()**. Надеемся, стало понятно почему **BCAR::BCAR()** определяется с пустым телом. Потом в конструкторе конкретного объекта вызывается **MemoryManager.alloc(int)**, благодаря чему менеджер памяти получает информацию о точном размере объекта и соответствующим образом правит свои структуры.

Уничтожение объектов примитивно, ибо всей необходимой информацией **MemoryManager** располагает:

```
void BCAR::operator delete( void *p ) {
    MemoryManager.free( (BCAR *)p, ((BCAR *)p)->size() );
}
```

Переносимость этой конструкции очень высока, хотя может понадобиться некоторая правка для весьма экзотических машин. Факты же таковы, что она используется в трёх очень крупных мировых центрах на четырёх аппаратных платформах и пяти операционках.

Но это ещё не всё. Как особо дотошные могли заметить — здесь присутствует виртуальность конструктора, но в любом случае объект конкретного класса всё равно имеет фиксированный размер. А вот объектов одного класса, но разного размера нет. До относительно недавнего времени нас это вполне устраивало, пока не появились некоторые требования, в результате которых нам пришлось сделать и это. Для этого у нас есть два (по меньшей мере) способа. Один — переносимый, но неэстетичный, а второй — непереносимый, но из **common practice**. Эта самая **common practice** состоит в помещении последним членом класса конструкции вида **unsigned char storage[1]** в расчёте на то, что это будет действительно последним байтом во внутреннем представлении объекта и туда можно записать не байт, а сколько надо. Стандарт этого вовсе не гарантирует, но практика распространения нашего детища показала, что для применяемых нами компиляторов оно именно так и есть. И оно работает. Чуть-чуть поправим наши объекты:

```
class Blood: public BCAR {

friend BCAR;

private:
    int bodySize;
    int size() { return sizeof( Blood ) + bodySize; }
    int getSize( const char * );
    sturct BloodAnalysisBody {
        // тут его поля
    } *body;
    Blood( const unsigned char *data ): BCAR() {
        body = (BloodAnalysisBody *) bodyStorage;
```

```
        bodySize = getSize( data );
        ::memcpy( bodyStorage, data + sizeof( header ),
bodySize );
        MemoryManager.alloc( size() );
    }
    unsigned char bodyStorage[ 1 ];
}
```

Бороться с данными далее придётся через **body->**, но сейчас мы не об этом...

Однако вспомним, что менеджер памяти у нас «свой в доску», и мы можем обойтись действительно переносимой конструкцией. Тело анализа достаточно разместить сразу за самим объектом, статический размер которого нам всегда известен. Ещё чуть-чуть правим:

```
class Blood: public BCAR {

friend BCAR;

private:
    int bodySize;
    int size() { return sizeof( Blood ) + bodySize; }
    int getSize( const unsigned char * );
    struct BloodAnalysisBody {
       // тут его поля
    } *body;
    Blood( const unsigned char *data ): BCAR() {
       body = (BloodAnalysisBody *) ((unsigned char *)this
   + sizeof( Blood ));
       bodySize = getSize( data );
       ::memcpy( body, data + sizeof( header ), bodySize );
       MemoryManager.alloc( size() );
    }
}
```

Данные гарантированно ложатся сразу за объектом в памяти, которую нам дал **MemoryManager** (а он, напомним, даёт нам всегда максимум из того, что имеет), а затем **alloc** соответствующим образом всё подправит.

Чтение исходных текстов

Хочется сразу же дать некоторые определения. Существует программирование для какой-то операционной системы. Программист, который пишет «под Unix», «под Windows», «под DOS», это такой человек, который знает (или догадывается) зачем нужен какой-либо системный вызов, умеет написать драйвер устройства и пытался дизассемблировать код ядра (это не относится к Unix — программист под Unix обычно пытается внести свои изменения в исходный текст ядра и посмотреть что получится).

Существует программирование на каком-то языке программирования. Программист, претендующий на то, что он является программистом «на Бейсике», «на Паскале», «на Си» или «на C++» может вообще ничего не знать о существовании конкретных операционных систем и архитектур, но обязан при этом знать тонкости своего языка программирования, знать о том, какие конструкции языка для чего эффективнее и т.д. и т.п.

Понятно, что «программирование на языке» и «программирование под архитектуру» могут вполне пересекаться. Программист на Си под Unix, например. Тем не менее, в особенности среди новичков, часто встречается подмена этих понятий. Стандартный вопрос новичка: «я только начал изучать C++, подскажите пожалуйста, как создать окно произвольной формы?». В общем, нельзя, наверное, изучать сразу же операционную систему и язык программирования. Вопрос же «про окно» правомерен, наверное, только для Java, где работа с окнами находится в стандартной библиотеке языка. Но у C++ своя специфика, потому что даже задав вопрос: «Как напечатать на экране строку Hello, world!?» можно напороться на раздраженное «Покажи где у C++ экран?».

Тем не менее, возвратимся к исходным текстам. Начинающие программисты обычно любят устраивать у себя разнообразные коллекции исходных текстов. Т.е., с десяток эмуляторов терминалов, пятнадцать библиотек пользовательского интерфейса, полсотни DirectX и OpenGL программ «на Си». Кстати сказать, программирование с использованием OpenGL тоже можно отнести к отдельному классу, который ортогонален классам «операционная система» и «язык программирования». Почему-то люди упорно считают, что набрав большое количество

различных программ малой, средней и большой тяжести, они решат себе много проблем, связанных с программированием. Это не так — совершенно не понятно, чем на стадии обучения программированию может помочь исходник Quake.

Этому есть простое объяснение. Читать исходные тексты чужих программ (да и своих, в принципе, тоже) очень нелегко. Умение «читать» программы само по себе признак высокого мастерства. Смотреть же на текст программ как на примеры использования чего-либо тоже нельзя, потому что в реальных программах есть очень много конкретных деталей, связанных со спецификой проекта, авторским подходом к решению некоторых задач и просто стилем программирования, а это очень сильно загораживает действительно существенные идеи.

Кроме того, при чтении «исходников», очень часто «программирование на языке» незаметно заменяется «программированием под ОС». Ведь всегда хочется сделать что-то красивое, чем можно удивить родителей или знакомых? А еще хочется сделать так, чтобы «как в Explorer» — трудно забыть тот бум на компоненты flat buttons для Delphi/C++ Builder, когда только появился Internet Explorer 3.0. Это было что-то страшное, таких компонент появилось просто дикое количество, а сколько программ сразу же появилось с их присутствием в интерфейсе...

Изменять текст существующей программы тоже очень сложно. Дополнить ее какой-то новой возможностью, которая не ставилась в расчет первоначально, сложно вдвойне. Читать текст, который подвергался таким «изменениям» уже не просто сложно — практически невозможно. Именно для этого люди придумали модульное программирование — для того, чтобы сузить, как только это возможно, степень зависимости между собой частей программы, которые пишутся различными программистами, или могут потребоваться в дальнейшем.

Чтение исходных текстов полезно, но уже потом, когда проблем с языком не будет (а до этого момента можно только перенять чужие ошибки), их коллекционирование никак не может помочь в начальном изучении языка программирования. Приемы, которые применяются разработчиками, значительно лучше воспринимаются когда они расписаны без излишних деталей и с большим количеством комментариев, для этого можно посоветовать книгу Джеффа Элджера «C++».

Функция gets()

Функция **gets()**, входящая в состав стандартной библиотеки Си, имеет следующий прототип:

```
char* gets(char* s);
```

Это определение содержится в **stdio.h**. Функция предназначена для ввода строки символов из файла **stdin**. Она возвращает **s** если чтение прошло успешно и **NULL** в обратном случае.

При всей простоте и понятности, эта функция уникальна. Все дело в том, что более опасного вызова, чем этот, в стандартной библиотеке нет... почему это так, а также чем грозит использование **gets()**, мы как раз и попытаемся объяснить далее.

Вообще говоря, для тех, кто не знает, почему использование функции **gets()** так опасно, будет полезно посмотреть еще раз на ее прототип, и подумать.

Все дело в том, что для **gets()** нельзя, т.е. совершенно невозможно, задать ограничение на размер читаемой строки, во всяком случае, в пределах стандартной библиотеки. Это крайне опасно, потому что тогда при работе с вашей программой могут возникать различные сбои при обычном вводе строк пользователями. Т.е., например:

```
char name[10];

// ...

puts("Enter you name:");
gets(name);
```

Если у пользователя будет имя больше, чем 9 символов, например, 10, то по адресу (name + 10) будет записан 0. Что там на самом деле находится, другие данные или просто незанятое место (возникшее, например, из-за того, что компилятор соответствующим образом выровнял данные), или этот адрес для программы недоступен, неизвестно.

Все эти ситуации ничего хорошего не сулят. Порча собственных данных означает то, что программа выдаст неверные результаты, а почему это происходит понять будет крайне трудно — первым делом программист будет проверять ошибки в алгоритме и только в конце заметит, что произошло

переполнение внутреннего буфера. Надеемся, все знают как это происходит — несколько часов непрерывных «бдений» с отладчиком, а потом через день, «на свежую голову», выясняется что где-то был пропущен один символ...

Опять же, для программиста самым удобным будет моментальное аварийное прекращение работы программы в этом месте — тогда он сможет заменить **gets()** на что-нибудь более «порядочное».

У кого-то может возникнуть предложение просто взять и увеличить размер буфера. Но не надо забывать, что всегда можно ввести строку длиной, превышающий выделенный размер; если кто-то хочет возразить, что случаи имен длиной более чем, например, 1024 байта все еще редки, то перейдем к другому, несколько более интересному примеру возникающей проблемы при использовании **gets()**.

Для это просто подчеркнем контекст, в котором происходит чтение строки.

```
void foo()
{
  char name[10];

  // ...

  puts("Enter you name:");
  gets(name);

  // ...
}
```

Имеется в виду, что теперь **name** расположен в стеке. Надеемся, что читающие эти строки люди имеют представление о том, как обычно выполняется вызов функции. Грубо говоря, сначала в стек помещается адрес возврата, а потом в нем же размещается память под массив **name**. Так что теперь, когда функция **gets()** будет писать за пределами массива, она будет портить адрес возврата из функции **foo()**.

На самом деле, это значит, что кто-то может задать вашей программе любой адрес, по которому она начнет выполнять инструкции.

Немного отвлечемся, потому что это достаточно интересно. В операционной системе Unix есть возможность запускать программы, которые будут иметь привилегии пользователя, отличного от того, кто этот запуск произвел. Самый распространенный пример, это, конечно же, суперпользователь. Например, команды **ps** или **passwd** при запуске любым пользователем получают полномочия root'а. Сделано это потому, что копаться в чужой памяти (для **ps**) может только суперпользователь, так же как и вносить изменения в **/etc/passwd**. Понятно, что такие программы тщательнейшим образом проверяются на отсутствие ошибок — через них могут «утечь» полномочия к нехорошим «хакерам» (существуют и хорошие!). Размещение буфера в стеке некоторой функции, чей код выполняется с привилегиями другого пользователя, позволяет при переполнении этого буфера изменить на что-то осмысленное адрес возврата из функции. Как поместить по переданному адресу то, что требуется выполнить (например, запуск командного интерпретатора), это уже другой разговор и он не имеет прямого отношения к программированию на Си или C++.

Принципиально иное: отсутствие проверки на переполнение внутренних буферов очень серьезная проблема. Зачастую программисты ее игнорируют, считая что некоторого заданного размера хватит на все, но это не так. Лучше с самого начала позаботиться о необходимых проверках, чтобы потом не мучиться с решением внезапно возникающих проблем. Даже если вы не будете писать программ, к которым выдвигаются повышенные требования по устойчивости к взлому, все равно будет приятно осознавать, что некоторых неприятностей возможно удалось избежать. А от **gets()** избавиться совсем просто:

```
fgets(name, 10, stdin);
```

Использование функции **gets()** дает лишнюю возможность сбоя вашей программы, поэтому ее использование крайне не рекомендовано. Обычно вызов **gets()** с успехом заменяется **fgets()**.

Свойства

Когда только появилась Delphi (первой версии) и ее мало кто видел, а еще меньше людей пробовали использовать, то в основном сведения об этом продукте фирмы Borland были похожи на сплетни. Помнится такое высказывание: «Delphi

расширяет концепции объектно-ориентированного программирования за счет использования свойств».

Те, кто работал с C++ Builder, представляет себе, что такое его свойства. Кроме того, кто хоть как-то знаком с объектно-ориентированным анализом и проектированием, понимает — наличие в языке той или иной конструкции никак не влияет на используемые подходы. Мало того, префикс «**ОО**» обозначает не использование ключевых слов **class** или **property** в программах, а именно подход к решению задачи в целом (в смысле ее архитектурного решения). С этой точки зрения, будут использоваться свойства или методы **set** и **get**, совершенно без разницы — главное разграничить доступ.

Тем не менее, свойства позволяют использовать следующую конструкцию: описать функции для чтения и записи значения и связать их одним именем. Если обратиться к этому имени, то в разных контекстах будет использоваться соответственно либо функция для чтения значения, либо функция для его установки.

Перед тем, как мы перейдем к описанию реализации, хотелось бы высказаться по поводу удобства использования. Прямо скажем, программисты — люди. Очень разные. Одним нравится одно, другое это ненавидят... в частности, возможность переопределения операций в C++ часто подвергается нападкам, при этом одним из основных аргументов (в принципе, не лишенный смысла) является то, что при виде такого выражения:

```
a = b;
```

нельзя сразу же сказать, что произойдет. Потому что на оператор присваивания можно «повесить» все что угодно. Самый распространенный пример «неправильного» (в смысле, изменение исходных целей) использования операций являются потоки ввода-вывода, в которых операторы побитового сдвига выполняют роль **printf**. Примерно то же нарекание можно отнести и к использованию свойств.

Тем не менее, перейдем к описанию примера. Итак, нужно оформить такой объект, при помощи которого можно было бы делать примерно следующее:

```
class Test
{
protected:
  int p_a;
```

```
    int& setA(const int& x);
    int getA();
public:
    /* ... */ a;
};

// ...

Test t;
t.a = 10;       // Вызов Test::setA()
int r = t.a;   // Вызов Test::getA()
```

Естественный способ — использовать шаблоны. Например, вот так:

```
template<class A, class T,
    T& (A::*setter)(const T&),
    T (A::*getter)()
    >
class property
{
    // ...
};
```

Параметр **A** — класс, к которому принадлежат функции установки и чтения значения свойств (они передаются через аргументы шаблона **setter** и **getter**). Параметр **T** — тип самого свойства (например, **int** для предыдущего примера).

На запись, которая используется для описания указателей на функции, стоит обратить внимание — известно, что многие программисты на C++ и не догадываются о том, что можно получить и использовать адрес функции-члена класса. Строго говоря, функция-член ничем не отличается от обычной, за исключением того, что ей требуется один неявный аргумент, который передается ей для определения того объекта класса, для которого она применяется (это указатель **this**). Таким образом, получение адреса для нее происходит аналогично, а вот использование указателя требует указать, какой конкретно объект используется. Запись **A::*foo** говорит о том, что это указатель на член класса **A** и для его использования потребуется объект этого класса.

Теперь непосредственно весь класс целиком:

```
template<class A, class T,
  T& (A::*setter)(const T&),
  T (A::*getter)()
  >
class property
{
protected:
  A * ptr;
public:
  property(A* p) : ptr(p) { }

  const T& operator= (const T& set) const
  {
    assert(setter != 0);
    return (ptr->*setter)(set);
  }
  operator T() const
  {
    assert(getter != 0);
    return (ptr->*getter)();
  }
};
```

Мы внесли тела функций внутрь класса для краткости (на самом деле, общая рекомендация никогда не захламлять определения классов реализациями функций — если нужен **inline**, то лучше его явно указать у тела подпрограммы, чем делать невозможным для чтения исходный текст. Приходится иногда видеть исходные тексты, в которых определения классов занимают по тысяче строк из-за того, что все методы были описаны внутри класса (как в Java). Это очень неудобно читать.

Думаем, что идея предельно ясна. Использование указателей на функцию-член заключается как раз в строках вида:

```
(ptr->*f)();
```

Указатель внутри свойства — это, конечно же, нехорошо. Тем не менее, без него не обойтись — указатель на объект нельзя будет передать в объявлении класса, только при создании объекта. Т.е., в параметры шаблона его никак не затолкать.

Использовать класс **property** надо следующим образом:

```
class Test
{
  // ...
  property<Test, int, &Test::setA, &Test::getA> a;

  Test() : a(this) { }
};
```

Компиляторы g++, msvc и bcc32 последних версий спокойно восприняли такое издевательство над собой. Тем не менее, более старые варианты этих же компиляторов могут ругаться на то, что используется незаконченный класс в параметрах шаблона, не понимать того, что берется адрес функций и т.д. Честно говоря, кажется что стандарту это не противоречит.

Мы подошли к главному — зачем все это нужно. А вообще не нужно. Тяжело представить программиста, который решится на использование подобного шаблона — это же просто нонсенс. Опять же, не видно никаких преимуществ по сравнению с обычным вызовом нужных функций и видно только один недостаток — лишний указатель. Так что все, что здесь изложено, стоит рассматривать только как попытку продемонстрировать то, что можно получить при использовании шаблонов.

На самом деле, теоретики «расширения концепций» очень часто забывают то, за что иногда действительно стоит уважать свойства. Это возможность сохранить через них объект и восстановить его (т.е., создать некоторое подобие **persistent object**). В принципе, добавить такую функциональность можно и в шаблон выше. Как? Это уже другой вопрос...

Таким образом, свойства как расширение языка ничего принципиально нового в него не добавляют. В принципе, возможна их реализация средствами самого языка, но использовать такую реализацию бессмысленно. Догадываться же, что такой подход возможен — полезно.

Комментарии

Плохое комментирование исходных текстов является одним из самых тяжелых заболеваний программ. Причем программисты

зачастую путают «хорошее» комментирование и «многословное». Согласитесь, комментарии вида:

```
i = 10; // Присваиваем значение 10 переменной i
```

выглядят диковато. Тем не менее, их очень просто расставлять и, поэтому, этим часто злоупотребляют. Хороший комментарий не должен находится внутри основного текста подпрограммы. Комментарий должен располагаться перед заголовком функции; пояснять что и как делает подпрограмма, какие условия накладываются на входные данные и что от нее можно ожидать; визуально отделять тела функций друг от друга. Потому что при просмотре текста программы зачастую незаметно, где заканчивается одна и начинается другая подпрограмма. Желательно оформлять такие комментарии подобным образом:

```
/*********************************************
 * function
 */
void function()
{
}
```

Этот комментарий можно использовать в любом случае, даже если из названия подпрограммы понятно, что она делает — продублировать ее название не тяжелый труд. Единственное, из опыта следует, что не надо переводить название функции на русский язык; если уж писать комментарий, то он должен что-то добавлять к имеющейся информации. А так видно, что комментарий выполняет декоративную роль.

Чем короче функция, тем лучше. Законченный кусочек программы, который и оформлен в виде независимого кода, значительно легче воспринимается, чем если бы он был внутри другой функции. Кроме того, всегда есть такая возможность, что этот коротенький кусочек потребуется в другом месте, а когда это требуется, программисты обычно поступают методом cut&paste, результаты которого очень трудно поддаются изменениям.

Комментарии внутри тела функции должны быть только в тех местах, на которые обязательно надо обратить внимание. Это не значит, что надо расписывать алгоритм, реализуемый функцией, по строкам функции. Не надо считать того, кто будет читать ваш текст, за идиота, который не сможет самостоятельно сопоставить ваши действия с имеющимся алгоритмом. Алгоритм

должен описываться перед заголовком в том самом большом комментарии, или должна быть дана ссылка на книгу, в которой этот алгоритм расписан.

Комментарии внутри тела подпрограммы могут появляться только в том случае, если ее тело все-таки стало длинным. Такое случается, когда, например, пишется подпрограмма для разбора выражений, там от этого никуда особенно не уйдешь. Комментарии внутри таких подпрограмм, поясняющие действие какого-либо блока, не должны состоять из одной строки, а обязательно занимать несколько строчек для того, чтобы на них обращали внимание. Обычно это выглядит следующим образом:

```
{
  /*
   * Комментарий
   */
}
```

Тогда он смотрится не как обычный текст, а именно как нужное пояснение.

Остальной текст, который желательно не комментировать, должен быть понятным. Названия переменных должны отображать их сущность и, по возможности, быть выполнены в едином стиле. Не надо считать, что короткое имя лучше чем длинное; если из названия короткого не следует сразу его смысл, то это имя следует изменить на более длинное. Кроме того, для C++ обычно используют области видимости для создания более понятных имен. Например, **dictionary::Creator** и **index::Creator**: внутри области видимости можно использовать просто **Creator** (что тоже достаточно удобно, потому что в коде, который имеет отношение к словарю и так ясно, какой **Creator** может быть без префикса), а снаружи используется нужный префикс, по которому смысл имени становится понятным.

Кроме того, должны быть очень подробно прокомментированы интерфейсы. Иерархии классов должны быть широкими, а не глубокими. Все дело в подходе: вы описываете интерфейс, которому должны удовлетворять объекты, а после этого реализуете конкретные виды этих объектов. Обычно все, что видит пользователь — это только определение базового класса такой «широкой» иерархии классов, поэтому оно должно быть максимально понятно для него. Кстати, именно для

возврата объектов, удовлетворяющих определенным интерфейсам, используют «умные» указатели.

Еще хорошо бы снабдить каждый заголовочный файл кратким комментарием, поясняющим то, что в нем находится. Файлы реализации обычно смотрятся потом, когда по заголовочным файлам становится понятно, что и где находится, поэтому там такие комментарии возникают по мере необходимости.

Таким образом, комментарии не должны затрагивать конкретно исходный текст программы, они все являются декларативными и должны давать возможность читателю понять суть работы программы не вдаваясь в детали. Все необходимые детали становятся понятными при внимательном изучении кода, поэтому комментарии рядом с кодом будут только отвлекать.

Следовательно, надо предоставить читателю возможность отвлеченного ознакомления с кодом. Под этим подразумевается возможность удобного листания комментариев или распечаток программной документации. Подобную возможность обеспечивают программы автоматизации создания программной документации. Таких программ достаточно много, для Java, например, существует JavaDoc, для C++ — doc++ и doxygen. Все они позволяют сделать по специального вида комментариям качественную документацию с большим количеством перекрестных ссылок и индексов.

Вообще, хотелось бы немного отклониться от основной темы и пофилософствовать. Хороший комментарий сам по себе не появляется. Он является плодом тщательнейшей проработки алгоритма подпрограммы, анализа ситуации и прочее. Поэтому когда становится тяжело комментировать то, что вы хотите сделать, то это означает, скорее всего, то, что вы сами еще плохо сознаете, что хотите сделать. Из этого логичным образом вытекает то, что комментарии должны быть готовы до того, как вы начали программировать.

Это предполагает, что вы сначала пишите документацию, а потом по ней строите исходный текст. Такой подход называется «литературным программированием» и автором данной концепции является сам Дональд Кнут (тот самый, который написал «Искусство программирования» и сделал TeX). У него даже есть программа, которая автоматизирует этот процесс, она

называется Web. Изначально она была разработана для Pascal'я, но потом появились варианты для других языков. Например, CWeb — это Web для языка Си.

Используя Web вы описываете работу вашей программы, сначала самым общим образом. Затем описываете какие-то более специфические вещи и т.д. Кусочки кода появляются на самых глубоких уровнях вложенности этих описаний, что позволяет говорить о том, что и читатель, и вы, дойдете до реализации только после того, как поймете действие программы. Сам Web состоит из двух программ: для создания программной документации (получаются очень красивые отчеты) и создания исходного текста на целевом языке программирования. Кстати сказать, TeX написан на Web'e, а документацию Web делает с использованием TeX'a... как вы думаете, что из них раньше появилось?

Web в основном применятся программистами, которые пишут сложные в алгоритмическом отношении программы. Сам факт присутствия документации, полученной из Web-исходника, упрощает ручное доказательство программ (если такое проводится). Тем не менее, общий подход к программированию должен быть именно такой: сначала думать, потом делать. При этом не надо мерить работу программиста по количеству строк им написанных.

Несмотря на то, что использование Web'a для большинства «совсем» прикладных задач несколько неразумно (хотя вполне возможно, кто мешает хорошо программировать?), существуют более простые реализации той же идеи.

Рассмотрим **doxygen** в качестве примера.

```
/**
 * \bref Краткое описание.
 *
 * Этот класс служит для демонстрации
 * возможностей автодокументации.
 */
class AutoDoc
{
public:
    int foo() const; //!< Просто функция. Возвращает ноль.
```

```
    /**
     * \brief Другая просто функция.
     *
     * Назначение непонятно: возвращает ноль. foo() -
     * более безопасная реализация возврата нуля.
     *
     * \param ignored - игнорируется.
     *
     * \warning может отформатировать жесткий диск.
     *
     * \note форматирование происходит только по пятницам.
     *
     * \see foo().
     */
    int bar(int ignored);
};
```

Комментарии для **doxygen** записываются в специальном формате. Они начинаются с «/**», «/*!» или «//!<». Весь подобный текст **doxygen** будет использовать в создаваемой документации. Он автоматически распознает имена функций или классов и генерирует ссылки на них в документации (если они присутствуют в исходном тексте). Внутри комментариев существует возможность использовать различные стилевые команды (пример достаточно наглядно демонстрирует некоторые из них), которые позволяют более тщательным образом структурировать документацию.

doxygen создает документацию в форматах:

- html;
- LaTeX;
- RTF;
- man.

Кроме того, из документации в этих форматах, можно (используя сторонние утилиты) получить документацию в виде MS HTML Help (наподобие MSDN), PDF (через LaTeX).

В общем использовать его просто, а результат получается очень хороший.

Кроме того (раз уж зашла об этом речь), существует такая программа, называется pcGRASP, которая позволяет

«разрисовать» исходный текст. В чем это заключается: все видели красивые и очень бесполезные блок-схемы. pcGRASP делает кое-что в этом духе: к исходному тексту он добавляет в левое поле разные линии (характеризующие уровень вложенности), ромбики (соответствующие операторам выбора), стрелочки (при переходе с одного уровня вложенности на другой, например, **return**) и т.д. Самое приятное, так это распечатки, полученные таким образом. Учитывая то, что **indent** (отступы) pcGRASP делает сам (не ориентируясь на то, как оно было), это делает подобные распечатки исключительно ценными.

В заключении, еще раз отметим, что комментарии надо писать так, чтобы потом самому было бы приятно их читать. Никакого распускания соплей в исходном тексте — тот, кто будет читать его, прекрасно знает что делает операция присваивания и нечего ему это объяснять.

Красиво оформленная программная документация является большим плюсом, по крайней мере потому, что люди, принимающие исходный текст, будут рады увидеть текст с гиперссылками. Тем более, что не составляет труда подготовить исходный текст к обработке утилитой наподобие **doxygen**.

Веб-программирование

Популярный нынче термин «веб-программирование» обычно подразумевает под собой программирование, в лучшем случае, на perl, в худшем — на PHP, в совсем тяжелом — на JavaScript. Ничуть не пытаемся обидеть людей, которые этим занимаются, просто смешно выделять «программирование на Perl» в отдельную нишу, прежде всего потому что специфика его использования отнюдь не в «веб-программировании».

Кроме того, вообще тяжело понять, чем принципиально отличается создание CGI-программ от «просто программ», кроме использования специализированного интерфейса для получения данных.

Тем не менее, «веб-программирование» действительно существует. Заключается оно не в генерации на лету HTML-страничек по шаблонам на основании информации из базы данных, потому что это относится именно к использованию БД. «Веб-программирование» — это работа с сетевыми

протоколами передачи данных, да и сетями вообще. Более точно, это «программирование для сетей, основанных на TCP/IP». А подобное программирование подразумевает прежде всего передачу данных, а не их обработку — скажите, что и куда передает CGI-программа по сети? Данные передает веб-сервер, а CGI используется как метод расширения возможностей сервера.

Настоящее веб-программирование — это программирование веб-серверов и веб-клиентов. Т.е., написать Apache, Internet Explorer или lynx — это веб-программирование.

Некоторые могут сказать, что мы слишком строго подошли к программированию CGI-приложений и, если копать дальше, то веб-программирование в том смысле, в котором мы его определили только что, является всего-навсего обработкой устройств ввода-вывода (к коим относится в одинаковой степени и сетевая карта, и клавиатура). Ну... да, это будет законный упрек. Только мы не собираемся так далеко заходить, просто хочется точнее определить термин «веб-программирование». Все дело в том, что генерация страниц «на лету» подразумевает то, что они будут отдаваться по сети, но это совершенно не обязательно. Неужели что-то принципиальным образом изменится в CGI-приложении, если его результаты будут сохраняться на жесткий диск? А внутри того, что подсовывается на вход PHP-интерпретатору? Ничего не изменится. Вообще. Для них главным является корректная установка нужных переменных среды окружения, а тот факт, подключен компьютер к сети, или нет, их не волнует.

Проблему передачи или получения данных через TCP, конечно же, тоже можно аналогичным образом развернуть и сказать, что с появлением интерфейса сокетов (BSD sockets) передача данных на расстояние ничем принципиально не отличается от работы с файлами на локальном диске.

В принципе это так. Потому что, если откинуть использование функций, связанных с инициализацией сокетов, в остальном с ними можно общаться как с традиционными файловыми дескрипторами.

Все это хорошо, но до некоторой поры. Все дело в том, что два компьютера могут находится рядом и быть соединены всего-лишь одним кабелем, а могут отстоять друг от друга на тысячи километров. И хотя скорость передачи данных в пределах

одного кабеля очень большая, но при использовании различных устройств для соединения кабелей друг с другом, происходит замедление передачи данных и чем «умнее» будет устройство, тем медленнее через него будут передаваться данные.

Это первое отличие. При работе с файловой системой программист никогда не думает о том, с какой скоростью у него считается файл или запишется (точнее, думает, но реже), буферизация обычно уже реализована на уровне операционной системы или библиотеки языка программирования, а при работе с сетью задержки могут быть очень велики и в это время программа будет ожидать прихода новых данных, вместо того, чтобы сделать что-либо полезное. Таким образом приходит необходимость разбивать программу на совокупность потоков и программирование превращается в ад, потому что ничего хорошего это не принесет, только лишние проблемы. Связано это с тем, что разделение программы на несколько одновременно выполняющихся потоков требует очень большой внимательности и поэтому чревато серьезными ошибками. А перевод существующей однопоточной программы в многопоточную, вообще занятие неблагодарное.

Второе отличие, в принципе, вытекает из первого, но стоит несколько отдельно, потому что это требование более жесткое. Все дело в том, что передача данных обычно не ограничивается одним файлом или одним соединением. Обычно сервер обслуживает одновременно несколько клиентов или клиент одновременно пытается получить доступ одновременно к нескольким ресурсам (они так выкачиваются быстрее, чем по одиночке). Тем самым приходиться открывать одновременно несколько сокетов и пытаться одновременно обработать их все.

Для подобной работы, в принципе, не требуется реальная многопоточность, существует такой вызов (или подобный ему в других операционных системах), называемый **select()**, который переводит программу в режим ожидания до тех пор, пока один (или несколько) файловых дескрипторов не станет доступным для чтения или записи. Это позволяет без введения нескольких потоков обрабатывать данные из нескольких соединений по мере их прихода.

Третье отличие, по сути, относится к серверным приложениям и выражается в повышенных требованиях к

производительности. Все дело в том, что когда один и тот же интернет-ресурс запрашивается большим количеством пользователей одновременно, то становится очень важна скорость реакции на него. В принципе, тут, если использовать CGI-приложения, на них тоже будет распространяться это требование, но в таких ситуациях обычно вставляют нужные обработчики непосредственно в сервер для повышения производительности.

Для облегчения написания веб-приложений (именно веб-приложений!) на языках Си и С++ была написана библиотека **libwww**. Она реализует псевдопотоковую событийную модель программы.

Под псевдопотоками понимается как раз использование **select()** для обработки большого количества запросов, а не введение для каждого из них настоящего потока операционной системы. Событийная модель предполагает то, что приложение запускает цикл обработки событий, поступающих от **libwww**, и в дальнейшем работа программы основывается на выполнении предоставленных обработчиков событий из цикла.

Приложение в этом случае управляется поступающими или инициированными запросами. Кроме этого псевдопараллельного «фетчера» (от слова fetch), в **libwww** присутствует большое количество готовых обработчиков, которые помогают решать типовые задачи, появляющиеся вместе с «веб-программированием». Например, существует набор «переводчиков», которые позволяют, например, из потока «**text/html**» сделать поток «**text/present**» (это в терминологии **libwww**, «**present**» означает вид, который будет представлен пользователю). Для этого используется собственный html-парсер (основанный на sgml-парсере с html dtd).

Таким образом, типичное веб-приложение, написанное при помощи **libwww**, выглядит как набор подпрограмм, обрабатывающие различные сообщения. Это позволяет написать как клиентские, так и серверные программы.

В качестве примера использования **libwww**, можно вспомнить текстовый браузер **lynx**, который написан при помощи этой библиотеки.

Впрочем, не будем распространяться долго об архитектуре **libwww**, желающие могут посмотреть документацию. Теперь нам хотелось бы пройтись по недостаткам.

libwww со временем становится все сложнее и сложнее по причине увеличивающегося набора стандартных обработчиков. Зачастую, достаточно простая задача, но которая не предусмотрена изначально как типовая, может потребовать нескольких часов изучения документации в поисках того, какие обработчики и где должны для этого присутствовать.

Документация на библиотеку не отличается подробностью и, как следствие, понятностью; она сводится к двум-трем строкам комментариев к программным вызовам, при этом местами документация просто устарела. Совет: если вы используете **libwww**, то обязательно смотрите исходный текст, по нему станет многое понятно. В качестве примера, можно привести такую вещь. Для того, чтобы отследить контекст разбора HTML, вводится специальный объект типа **HText**, который используется в обработчиках событий от парсера в качестве «внешней памяти». Это как указатель **this** в C++, т.е. реализация объектов на языке Си. Тип **HText** целиком предоставляется пользователем, вместе с callback-функциями создания и удаления. Прежде чем его использовать, надо зарегистрировать эти функции (причем обязательно парой, т.е. надо обязательно указать и функцию создания объекта, и функцию разрушения), которые, если следовать документации, будут вызваны, соответственно, для создания и удаления объекта при начале и конце разбора соответственно.

Это все пока что «хорошо». «Плохо» — функция разрушения не вызывается никогда. В частности, в документации этого нет, но видно по исходному тексту, где в том месте, где должен был находится вызов удаляющей функции, находится комментарий, в котором написано, что забота об удалении ложится на плечи приложения, а не библиотеки **libwww**. Кто при этом мешает вызвать переданную функцию, не понятно. Кстати сказать, примеры использования объекта **HText** в поставляемых с **libwww** приложениях из каталога **Examples**, рассчитаны на то, что функция удаления будет вызвана.

И это не единственное забавное место в библиотеке. Все это решаемо, конечно, но лучше если вы будете сразу же смотреть

исходный текст **libwww**, по нему значительно больше можно понять, чем по предоставленной документации.

Итак, использование **libwww** позволяет быстро написать типичное web-приложение, например, «робота», который выкачивает определенные документы и каким-то образом их анализирует. Тем не менее, использовать **libwww** в своих программах не всегда просто из-за проблем с документацией на нее.

Ошибки работы с памятью

Когда программа становится внушительной по своему содержанию (то есть, не по количеству строчек, а по непонятности внутренних связей), то ее поведение становится похожим на поведение настоящего живого существа. Такое же непредсказуемое... впрочем, кое что все-таки предсказать можно: работать оно не будет. Во всяком случае, сразу.

Программирование на Си и C++ дает возможность допускать такие ошибки, поиск которых озадачил бы самого Шерлока Холмса. Вообще говоря, чем загадочнее ведет себя программа, тем проще в ней допущена ошибка. А искать простые ошибки сложнее всего, как это ни странно; все потому, что сложная ошибка обычно приводит к каким-то принципиальным неточностям в работе программы, а ошибка простая либо превращает всю работу в «бред пьяного программиста», либо всегда приводит к одному и тому же: *segmentation fault*.

И зря говорят, что если ваша программа выдала фразу **core dumped**, то ошибку найти очень просто: это, мол, всего лишь обращение по неверному указателю, например, нулевому. Обращение-то, конечно же, есть, но вот почему в указателе появилось неверное значение? Откуда оно взялось? Зачастую на этот вопрос не так просто ответить.

В Java исключены указатели именно потому, что работа с ними является основным источником ошибок программистов. При этом отсутствие инициализации является одним из самых простых и легко отлавливаемых вариантов ошибок.

Самые трудные ошибки появляются, как правило, тогда, когда в программе постоянно идут процессы выделения и удаления памяти. То есть, в короткие промежутки времени

появляются объекты и уничтожаются. В этом случае, если где-нибудь что-нибудь некорректно «указать», то «core dumped», вполне вероятно, появится не сразу, а лишь через некоторое время. Все дело в том, что ошибки с указателями проявляются обычно в двух случаях: работа с несуществующим указателем и выход за пределы массива (тоже в конечном итоге сводится к несуществующему указателю, но несколько чаще встречается).

Загадки, возникающие при удалении незанятой памяти, одни из самых трудных. Выход за границы массива, пожалуй, еще сложнее.

Представьте себе: вы выделили некоторый буфер и в него что-то записываете, какие-то промежуточные данные. Это критическое по времени место, поэтому тут быть не может никаких проверок и, ко всему прочему, вы уверены в том, что исходного размера буфера хватит на все, что в него будут писать. Не хотелось бы торопиться с подобными утверждениями: а почему, собственно, вы так в этом уверены? И вообще, а вы уверены в том, что правильно вычислили этот самый размер буфера?

Ответы на эти вопросы должны у вас быть. Мало того, они должны находиться в комментариях рядом с вычислением размера буфера и его заполнением, чтобы потом не гадать, чем руководствовался автор, когда написал:

```
char buf[100];
```

Что он хотел сказать? Откуда взялось число 100? Совершенно непонятно.

Теперь о том, почему важно не ошибиться с размерами. Представьте себе, что вы вышли за пределы массива. Там может «ничего не быть», т.е. этот адрес не принадлежит программе и тогда в нормальной операционной системе вы получите соответствующее «матерное» выражение. А если там что-то было?

Самый простой случай — если там были просто данные. Например, какое-нибудь число. Тогда ошибка, по крайней мере, будет видна почти сразу... а если там находился другой указатель? Тогда у вас получается наведенная ошибка очень высокой сложности. Потому что вы будете очень долго искать то место, где вы забыли нужным образом проинициализировать этот указатель...

Мало того, подобные «наведенные» ошибки вполне могут вести себя по-разному не только на разных тестах, но и на одинаковых.

А если еще программа «кормится» данными, которые поступают непрерывно... и еще она сделана таким образом, что реагирует на события, которые каким-то образом распределяются циклом обработки событий... тогда все будет совсем плохо. Отлаживать подобные программы очень сложно, тем более что зачастую, для того, чтобы получить замеченную ошибку повторно, может потребоваться несколько часов выполнения программы. И что делать в этих случаях?

Поиск таких ошибок более всего напоминает шаманские пляски с бубном около костра, не зря этот образ появился в программистском жаргоне. Потому что программист, измученный бдениями, начинает просто случайным образом «удалять» (закомментировав некоторую область, или набрав **#if 0 ... #endif**) блоки своей программы, чтобы посмотреть, в каком случае оно будет работать, а в каком — нет.

Это действительно напоминает шаманство, потому что иногда программист уже не верит в то, что, например, «от перестановки мест сумма слагаемых не меняется» и запросто может попытаться переставить и проверить результат... авось?

А вот теперь мы подошли к тому, что в шаманстве тоже можно выделить систему. Для этого достаточно осознать, что большинство загадочных ошибок происходят именно из-за манипуляций с указателями. Поэтому, вместо того чтобы переставлять местами строчки программы, можно просто попытаться для начала закомментировать в некоторых особенно опасных местах удаление выделенной памяти и посмотреть что получится.

Кстати сказать, отладка таких моментов требует (именно требует) наличия отладочной информации во всех используемых библиотеках, так будет легче работать. Так что, если есть возможность скомпилировать библиотеку с отладочной информацией, то так и надо делать — от лишнего можно будет избавиться потом.

Если загадки остались, то надо двинуться дальше и проверить индексацию массивов на корректность. В идеале, перед каждым обращением к массиву должна находиться

проверка инварианта относительно того, что индекс находиться в допустимых пределах. Такие проверки надо делать отключаемыми при помощи макросов **DEBUG/RELEASE** с тем, чтобы в окончательной версии эти дополнительные проверки не мешались бы (этим, в конце-концов, Си отличается от Java: хотим — проверяем, не хотим — не проверяем). В этом случае вы значительно быстрее сможете найти глупую ошибку (а ошибки вообще не бывают умными; но найденные — глупее оставшихся)).

На самом деле, в C++ очень удобно использовать для подобных проверок шаблонные типы данных. То есть, сделать тип «массив», в котором переопределить необходимые операции, снабдив каждую из них нужными проверками. Операции необходимо реализовать как **inline**, это позволит не потерять эффективность работы программы. В то же самое время, очень легко будет удалить все отладочные проверки или вставить новые. В общем, реализация своего собственного типа данных **Buffer** является очень полезной.

Кстати, раз уж зашла об этом речь, то абзац выше является еще одним свидетельством того, что C++ надо использовать «полностью» и никогда не писать на нем как на «усовершенствованном Си». Если вы предпочитаете писать на Си, то именно его и надо использовать. При помощи C++ те же задачи решаются совсем по другому.

Создание графиков с помощью ploticus

Есть такая программа, предназначенная для создания графиков различных видов из командной строки, называется **ploticus**. Программа сама по себе достаточно удобная — потому что иногда очень полезно автоматизировать генерацию различных графических отчетов, а тут без командной строки и вызова программ из скриптов не обойтись.

Нет, таких программ великое множество, но **ploticus** отличается от них очень удобным преимуществом: он «глупый». То есть, его можно, например, заставить разместить надпись на рисунке с точностью до пиксела... иногда это нужно.

Но разговор не об удобстве этой программы. Просто иногда требуется использовать **ploticus**, но при этом немного доработанный напильником.

Сначала немного лирики. **Ploticus**, для того, чтобы построить график, читает некоторый файл, в котором находится определение этого самого графика (скрипт, так сказать). Этот файл обладает очень простой грамматикой. Мало того, **ploticus** умеет организовывать программный канал (хотя, кто этого не умеет?) и читать данные оттуда, как результат выполнения другой программы.

Так вот о чтении этого файла мы и хотим немного рассказать. Итак, подпрограммы чтения данных в **ploticus** разбиты на некоторые логические блоки, исходя из структуры самого файла с данными. Ну это понятно и логично. Не особенно понятно другое: каждая подпрограмма (парсер) на вход воспринимает название файла, в котором находится содержимое. В итоге, основная подпрограмма сначала разбивает файл на блоки, содержимое этих файлов копирует (!) во временные файлы (!!), которые подсовывает на вход другим подпрограммам. Это, конечно, уже достаточно оригинально, хотя задумка автора ясна — он хотел сделать так, чтобы в этих местах на вход подпрограммам чтения данных можно было бы подсунуть имя программы, которая эти данные бы сгенерировала. Тем не менее, можно было бы сделать значительно красивее, чем создавать кучу временных текстовых файлов.

Эти «подпарсеры» реализованы... аналогичным образом. Т.е., автор не смущаясь разбивает подсунутые файлы еще на кусочки и записывает их в другие временные файлы, которые потом читает.

В принципе, все эти места как раз и требовали вмешательства напильника, потому что хотелось иметь программный интерфейс ко всему этому хозяйству и, желательно, чтобы данные не покидали оперативной памяти.

Все написанное выше уже смешно. Но кусочек кода, который приведен, может довести программиста до истерического смеха. Вот он (с купюрами):

```
/*
 * ...
 */
```

```
else if( stricmp( attr, "data" )==0 ) {
   FILE *tfp;
   sprintf( datafile, "%s_D", Tmpname );
   getmultiline( "data", lineval, fp, MAXBIGBUF, Bigbuf );
   tfp = fopen( datafile, "w" );
   if( tfp == NULL ) return( Eerr( 294, "Cannot open tmp
data file", datafile ));
   fprintf( tfp, "%s", Bigbuf );
   fclose( tfp );
   }
/*
 * ...
 */
```

Это как раз и есть выделение секции с данными и запись ее во временный файл. Вообще, использование **fprintf** с шаблоном **"%s"** уже смотрится очень оригинально, но то, что идет 80 строками ниже еще более необычно:

```
/*
 * ...
 */
if( standardinput || strcmp( datafile, "-" ) ==0 ) { /* a
file of "-" means read from stdin */
   dfp = stdin;
   goto PT1;
   }
if( strlen( datafile ) > 0 ) sprintf( command, "cat %s",
datafile );
if( strlen( command ) > 0 ) {
   dfp = popen( command, "r" );
   if( dfp == NULL ) {
         Skipout = 1;
         return( Eerr( 401, "Cannot open", command ) );
         }

   PT1:
/*
 * ...
 */
```

Обратите внимание на строчку:

```
if( strlen( datafile ) > 0 ) sprintf( command, "cat %s",
datafile );
```

и следующую за ней:

```
dfp = popen( command, "r" );
```

Честно говоря, это впечатляет. Очень впечатляет... при этом, совершено не понятно что мешало использовать обычный **fopen()** для этого (раз уж так хочется), раз уж есть строки вида:

```
dfp = stdin;
goto PT1;
```

В общем, дикость. Если кто-то не понял, то объясним то, что происходит, на пальцах: читается секция «**data**» и ее содержимое записывается в файл, название которого содержится в **datafile**. Потом, проверяется название этого файла, если оно равно «**-**», то это значит, что данные ожидаются со стандартного файла ввода, **stdin**. Если же нет, то проверяется длина строки, на которую указывает **datafile**. Если она ненулевая, то считается, что команда, результаты работы которой будут считаться за входные данные, это «**cat datafile**». Если же ненулевая длина у другого параметра, **command**, то его значение принимается за выполняемую команду. После всех этих манипуляций, открывается программный канал (**pipe**) при помощи **popen()**, результатом которой является обычный указатель на структуру **FILE** (при его помощи можно использовать обычные средства ввода-вывода).

Вам это не смешно?

Автоматизация и моторизация приложения

Программирование давно стало сплавом творчества и строительства, оставляя в прошлом сугубо научно-шаманскую окраску ремесла. И если такой переход уже сделан, то сейчас можно обозначить новый виток – ломание барьеров API и переход к более обобщенному подходу в проектировании, выход на новый уровень абстракции. Немало этому способствовал Интернет и его грандиозное творение – XML. Сегодня ключ к успеху приложения сплавляется из способности его создателей обеспечить максимальную совместимость со стандартами и в то же время масштабируемость. Придумано такое количество

различных технологий для связи приложений и повторного использования кода, что сегодня прикладные программы не могут жить без такой «поддержки». Под термином «автоматизация» понимается настоящее оживление приложений, придание им способности взаимодействовать с внешней средой, предоставление пользователю максимального эффекта в работе с приложениями. Не равняясь на такие гранды технической документации, как MSDN, тем не менее, хотим указать на путь, по которому сегодня проектируются современные приложения.

Автоматизация как есть

Автоматизация (Automation) была изначально создана как способ для приложений (таких как Word или Excel) предоставлять свою функциональность другим приложениям, включая скрипт-языки. Основная идея заключалась в том, чтобы обеспечить наиболее удобный режим доступа к внутренним объектам, свойствам и методам приложения, не нуждаясь при этом в многочисленных «хедерах» и библиотеках.

Вообще, можно выделить два вида автоматизации – внешнюю и внутреннюю. Внешняя автоматизация – это работа сторонних приложений с объектной моделью вашей программы, а внутренняя – это когда сама программа предоставляет пользователю возможность работы со своей объектной структурой через скрипты. Комбинирование первого и второго вида представляется наиболее масштабируемым решением и именно о нем пойдет речь.

Для начала обратим внимание на самое дно – интерфейсы COM. Если термин «интерфейс» в этом контексте вам ничего не говорит, то представьте себе абстрактный класс без реализации – это и есть интерфейс. Реальные объекты наследуются от интерфейсов. Компоненты, наследующиеся от интерфейса IUnknown, называются COM-объектами. Этот интерфейс содержит методы подсчета ссылок и получения других интерфейсов объекта.

Автоматизация базируется на интерфейсе IDispatch, наследующегося от IUnknown. IDispatch позволяет запускать методы и обращаться к свойствам вашего объекта через их символьные имена. Интерфейс имеет немного методов, которые являются тем не менее довольно сложными в реализации. К счастью, существует множество шаблонных классов,

предлагающих функциональность интерфейса IDispatch, поэтому для создания объекта, готового к автоматизации, необходимо всего лишь несколько раз щелкнуть мышкой в ClassWizard Visual C++.

Что касается способа доступа и динамического создания ваших внутренних **dispatch** объектов, то тут тут тоже все довольно просто – данные об объекте хранятся в реестре под специальным кодовым именем, которое называется **ProgId**. Например, **progid** программы Excel – **Excel.Application**. Создать в любой процедуре на VBScript достаточно легко – надо только вызвать функцию **CreateObject**, в которую передать нужный **ProgID**. Функция вернет указатель на созданный объект.

MFC

В MFC существует специальный класс, под названием **CCmdTarget**. Наследуя свои классы от **CCmdTarget**, вы можете обеспечить для них необходимую функциональность в **dispatch** виде – как раз как ее понимают скрипты. При создании нового класса в **ClassWizard** (**View** ⇨ **ClassWizard** ⇨ **Add Class** ⇨ **New**), наследуемого от **CCmdTarget**, просто щелкните на кнопке **Automation** или **Creatable by ID**, чтобы обеспечить возможность создания экземпляра объекта по его **ProgID**. Заметим, что для программ, реализующих внутреннюю автоматизацию, это не нужно. Для приложений, реализующих внешнюю и смешанную автоматизацию, это необходимо для «корневых» объектов.

COM-объекты можно создавать только динамически. Это связано с тем, что объект может использоваться несколькими приложениями одновременно, а значит, удаление объекта из памяти не может выполнить ни одно из них. Разумно предположить, что объект сам должен отвечать за свое удаление. Такой механизм реализован при помощи механизма ссылок (reference count). Когда приложение получает указатель на объект, он увеличивает свой внутренний счетчик ссылок, а когда приложение освобождает объект – счетчик ссылок уменьшается. При достижении счетчиком нуля, объект удаляет сам себя. Если наш объект был создан по **ProgID** другим приложением, то программа **CTestApp** (другими словами, Automation-Server) не завершится до тех пор, пока счетчик ссылок **CTestAutomatedClass** не станет равным нулю.

Создаваемые через **ProgID** COM-объекты, обычно являются Proxy-компонентами. Реально они не содержат никакой функциональности, но имеют доступ к приложению и его внутренним, не доступным извне, функциям. Хотя можно организовать все таким образом, чтобы всегда создавался только один COM-объект, а все остальные вызовы на создание возвращали указатели на него.

Метод интерфейса **CCmdTarget GetIDispatch()**, позволяет получить указатель на реализованный интерфейс **IDispatch**. В параметрах можно указать, нужно ли увеличивать счетчик ссылок или нет.

Оболочка из классов для COM

Программировать с использованием COM настолько трудно, что вы не должны даже пробовать это без MFC. Правильно или неправильно? Абсолютная чушь! Рекламируемые OLE и его преемник COM имеют элегантность гиппопотама, занимающегося фигурным катанием. Но размещение MFC на вершине COM подобно одеванию гиппопотама в клоунский костюм еще больших размеров.

Итак, что делать программисту, когда он столкнется с потребностью использовать возможности оболочки Windows, которые являются доступными только через интерфейсы COM?

Для начала, всякий раз, когда вы планируете использовать COM, вы должны сообщить системе, чтобы она инициализировала COM подсистему. Точно так же всякий раз, когда вы заканчиваете работу, вы должны сообщить системе, чтобы она выгрузила COM. Самый простой способ это сделать заключается в определении объекта, конструктор которого инициализирует COM, а деструктор выгружает ее. Самое лучшее место, для внедрения данного механизма — это объект **Controller**.

```
class Controller
{
public:
     Controller (HWND hwnd, CREATESTRUCT * pCreate);
     ~Controller ();
     // ...
private:
     UseCom     _comUser; // I'm a COM user
```

```
        Model        _model;
        View         _view;
        HINSTANCE    _hInst;
};
```

Этот способ гарантирует, что COM подсистема будет проинициализирована прежде, чем к ней будут сделаны любые обращения и что она будет освобождена после того, как программа осуществит свои разрушения (то есть, после того, как «Вид» и «Модель» будут разрушены).

Класс **UseCom** очень прост.

```
class UseCom
{
public:
        UseCom ()
        {
                HRESULT err = CoInitialize (0);
                if (err != S_OK)
throw "Couldn't initialize COM";
        }
        ~UseCom ()
        {
                CoUninitialize ();
        }
};
```

Пока не было слишком трудно, не так ли? Дело в том, что мы не коснулись главной мерзости COM программирования — подсчета ссылок. Вам должно быть известно, что каждый раз, когда вы получаете интерфейс, его счетчик ссылок увеличивается. И вам необходимо явно уменьшать его. И это становится более чем ужасным тогда, когда вы начинаете запрашивать интерфейсы, копировать их, передавать другим и т.д. Но ловите момент: мы знаем, как управляться с такими проблемами! Это называется управлением ресурсами. Мы никогда не должны касаться интерфейсов COM без инкапсуляции их в интеллектуальных указателях на интерфейсы. Ниже показано, как это работает.

```
template <class T>class SIfacePtr
{
public:
        ~SIfacePtr ()
        {
```

```
            Free ();
        }
        T * operator->() { return _p; }
        T const * operator->() const { return _p; }
        operator T const * () const { return _p; }
        T const & GetAccess () const { return *_p; }

    protected:
        SIfacePtr () : _p (0) {}
        void Free ()
        {
            if (_p != 0)
    _p->Release ();
            _p = 0;
        }

        T * _p;
    private:
        SIfacePtr (SIfacePtr const & p) {}
        void operator = (SIfacePtr const & p) {}
    };
```

Не волнуйте, что этот класс выглядит непригодным (потому что имеет защищенный конструктор). Мы никогда не будем использовать его непосредственно. Мы наследуем от него. Между прочим, это удобный прием: создайте класс с защищенным пустым конструктором, и осуществите наследование от него. Все наследующие классы должны обеспечивать их внутреннюю реализацию своими открытыми конструкторами. Поскольку вы можете иметь различные классы, наследующие от **SIfacePtr**, они будут отличаться по способу получения, по их конструкторам, рассматриваемым интерфейсам.

Закрытый фиктивный копирующий конструктор и оператор «=» не всегда необходимы, но они сохранят вам время, потраченное на отладку, если по ошибке вы передадите интеллектуальный интерфейс по значению вместо передачи по ссылке. Это больше невозможно. Вы можете освободить один и тот же интерфейс, дважды и это будет круто отражаться на подсчете ссылок COM. Верьте, это случается. Как это часто бывает, компилятор откажется передавать по значению объект, который имеет закрытую копию конструктора. Он также выдаст

ошибку, когда вы пробуете присвоить объект, который имеет закрытый оператор присваивания.

Оболочки API часто распределяет память, используя свои собственные специальные программы распределения. Это не было бы настолько плохо, если бы не предположение, что они ожидают от вас освобождения памяти с использованием той же самой программы распределения. Так, всякий раз, когда оболочка вручает нам такой сомнительный пакет, мы оборачиваем его в специальный интеллектуальный указатель.

```
template <class T>
class SShellPtr
{
public:
    ~SShellPtr ()
    {
        Free ();
        _malloc->Release ();
    }
    T * weak operator->() { return _p; }
    T const * operator->() const { return _p; }
    operator T const * () const { return _p; }
    T const & GetAccess () const { return *_p; }

protected:
    SShellPtr () : _p (0)
    {
        // Obtain malloc here, rather than
        // in the destructor.
        // Destructor must be fail-proof.
        // Revisit: Would static IMalloc * _shellMalloc
work?
        if (SHGetMalloc (& _malloc) == E_FAIL)
throw Exception "Couldn't obtain Shell Malloc";
    }
    void Free ()
    {
        if (_p != 0)
_malloc->Free (_p);
        _p = 0;
    }
```

```
    T * _p;
    IMalloc *  _malloc;
private:
    SShellPtr (SShellPtr const & p) {}
    void operator = (SShellPtr const & p) {}
};
```

Обратите внимание на использование приема: класс **SShellPtr** непосредственно не пригоден для использования. Вы должны наследовать от него подкласс и реализовать в нем соответствующий конструктор.

Обратите также внимание, нет уверенности, может ли **_shellMalloc** быть статическим элементом **SShellPtr**. Проблема состоит в том, что статические элементы инициализируются перед **WinMain**. Из-за этого вся COM система может оказаться неустойчивой. С другой стороны, документация говорит, что вы можете безопасно вызывать из другой API функции **CoGetMalloc** перед обращением к **CoInitialize**. Это не говорит о том, может ли **SHGetMalloc**, который делает почти то же самое, также вызываться в любое время в вашей программе. Подобно многим другим случаям, когда система ужасно разработана или задокументирована, только эксперимент может ответить на такие вопросы.

Между прочим, если вы нуждаетесь в интеллектуальном указателе, который использует специфическое распределение памяти для COM, то получите его, вызывая **CoGetMalloc**. Вы можете без опаски сделать этот **_malloc** статическим элементом и инициализировать его только один раз в вашей программе (ниже **SComMalloc:: GetMalloc** тоже статический):

```
IMalloc * SComMalloc::_malloc = SComMalloc::GetMalloc ();
IMalloc * SComMalloc::GetMalloc ()
{
    IMalloc * malloc = 0;
    if (CoGetMalloc (1, & malloc) == S_OK)
        return malloc;
    else
        return 0;
}
```

Это — все, что надо знать, чтобы начать использовать оболочку Windows и ее COM интерфейсы. Ниже приводится

пример. Оболочка Windows имеет понятие Рабочего стола, являющегося корнем «файловой» системы. Вы обращали внимание, как Windows приложения допускают пользователя, просматривают файловую систему, начинающуюся на рабочем столе? Этим способом вы можете, например, создавать файлы непосредственно на вашем рабочем столе, двигаться между дисководами, просматривать сетевой дисковод, и т.д. Это, в действительности, Распределенная Файловая система (PMDFS — poor man's Distributed File System). Как ваше приложение может получить доступ к PMDFS? Просто. В качестве примера напишем код, который позволит пользователю выбирать папку, просматривая PMDFS. Все, что мы должны сделать — это овладеть рабочим столом, позиционироваться относительно его, запустить встроенное окно просмотра и сформировать путь, который выбрал пользователь.

```
char path [MAX_PATH];
path [0] = '\0';
Desktop desktop;
ShPath browseRoot (desktop, unicodePath);
if (browseRoot.IsOK ())
{
    FolderBrowser browser (hwnd,
        browseRoot,
        BIF_RETURNONLYFSDIRS,
       "Select folder of your choice");
    if (folder.IsOK ())
    {
        strcpy (path, browser.GetPath ());
    }
}
```

Давайте, запустим объект **desktop**. Он использует интерфейс по имени **IShellFolder**. Обратите внимание, как мы приходим к Первому Правилу Захвата. Мы распределяем ресурсы в конструкторе, вызывая функцию API **SHGetDesktopFolder**. Интеллектуальный указатель интерфейса будет заботиться об управлении ресурсами (подсчет ссылок).

```
class Desktop: public SIfacePtr<IShellFolder>
{
public:
    Desktop ()
```

```
    {
        if (SHGetDesktopFolder (& _p) != NOERROR)
throw "SHGetDesktopFolder failed";
    }
};
```

Как только мы получили рабочий стол, мы должны создать специальный вид пути, который используется PMDFS. Класс **ShPath** инкапсулирует этот «путь». Он создан из правильного **Unicode** пути (используйте **mbstowcs**, чтобы преобразовать путь ASCII в Unicode: **int mbstowcs(wchar_t *wchar, const char *mbchar, size_t count)**). Результат преобразования — обобщенный путь относительно рабочего стола. Обратите внимание, что память для нового пути распределена оболочкой — мы инкапсулируем это в **SShellPtr**, чтобы быть уверенными в правильном освобождении.

```
class ShPath: public SShellPtr<ITEMIDLIST>
{
public:
  ShPath (SIfacePtr<IShellFolder> & folder, wchar_t * path)
    {
        ULONG lenParsed = 0;
        _hresult =
folder->ParseDisplayName (0, 0, path, & lenParsed, & _p, 0);
    }
    bool IsOK () const { return SUCCEEDED (_hresult); }
private:
    HRESULT _hresult;
};
```

Этот путь оболочки станет корнем, из которого окно просмотра начнет его взаимодействие с пользователем.

С точки зрения клиентского кода, окно просмотра — путь, выбранный пользователем. Именно поэтому он наследуется от **SShellPtr<ITEMIDLIST>**.

Между прочим, **ITEMIDLIST** — официальное имя для этого обобщенного пути.

```
class FolderBrowser: public SShellPtr<ITEMIDLIST>
{
public:
    FolderBrowser (
        HWND hwndOwner,
```

```
        SShellPtr<ITEMIDLIST> & root,
        UINT browseForWhat,
        char const *title);
    char const * GetDisplayName () { return _displayName; }
    char const * GetPath ()        { return _fullPath; }
    bool IsOK() const  { return _p != 0; };

private:
    char        _displayName [MAX_PATH];
    char        _fullPath [MAX_PATH];
    BROWSEINFO _browseInfo;
};

FolderBrowser::FolderBrowser (
    HWND hwndOwner,
    SShellPtr<ITEMIDLIST> & root,
    UINT browseForWhat,
    char const *title)
{
    _displayName [0] = '\0';
    _fullPath [0] = '\0';
    _browseInfo.hwndOwner = hwndOwner;
    _browseInfo.pidlRoot = root;
    _browseInfo.pszDisplayName = _displayName;
    _browseInfo.lpszTitle = title;
    _browseInfo.ulFlags = browseForWhat;
    _browseInfo.lpfn = 0;
    _browseInfo.lParam = 0;
    _browseInfo.iImage = 0;
    // Let the user do the browsing
    _p = SHBrowseForFolder (& _browseInfo);

    if (_p != 0)
        SHGetPathFromIDList (_p, _fullPath);
}
```

Вот так! Разве это не просто?

Как функции, не являющиеся методами, улучшают инкапсуляцию

Когда приходится инкапсулировать, то иногда лучше меньше, чем больше.

Если вы пишете функцию, которая может быть выполнена или как метод класса, или быть внешней по отношению к классу, вы должны предпочесть ее реализацию без использования метода. Такое решение увеличивает инкапсуляцию класса. Когда вы думаете об использовании инкапсуляции, вы должны думать том, чтобы не использовать методы.

При изучении проблемы определения функций, связанных с классом для заданного класса **C** и функции **f**, связанной с **C**, рассмотрим следующий алгоритм:

```
if (f необходимо быть виртуальной)
   сделайте f функцией-членом C;
else if (f - это operator>> или operator<<)
{
   сделайте f функцией - не членом;
   if (f необходим доступ к непубличным членам C)
      сделайте f другом C;
}
else if (в f надо преобразовывать тип его крайнего левого
аргумента)
{
   сделайте f функцией - не членом;
   if (f необходимо иметь доступ к непубличным членам C)
      сделайте f другом C;
}
else
   сделайте f функцией-членом C;
```

Этот алгоритм показывает, что функции должны быть методами даже тогда, когда они могли бы быть реализованы как не члены, которые использовали только открытый интерфейс класса C. Другими словами, если **f** могла бы быть реализована как функция-член (метод) или как функция не являющаяся не другом, не членом, действительно ее надо реализовать как метод класса? Это не то, то, что подразумевалось. Поэтому алгоритм был изменен:

```
if (f необходимо быть виртуальной)
   сделайте f функцией-членом C;
else if (f - это operator>> или operator<<)
{
   сделайте f функцией - не членом;
   if (f необходим доступ к непубличным членам C)
```

```
        сделайте f другом C;
}
else if (f необходимо преобразовывать тип его крайнего
левого аргумента)
{
    сделайте f функцией - не членом;
    if (f необходимо иметь доступ к непубличным членам C)
        сделайте f другом C;
}
else if (f может быть реализована через доступный интерфейс
класса)
    сделайте f функцией - не членом;
else
    сделайте f функцией-членом C;
```

Инкапсуляция не определяет вершину мира. Нет ничего такого, что могло бы возвысить инкапсуляцию. Она полезна только потому, что влияет на другие аспекты нашей программы, о которых мы заботимся. В частности, она обеспечивает гибкость программы и ее устойчивость к ошибкам. Посмотрите на эту структуру, чья реализация не является инкапсулированной:

```
struct Point {
    int x, y;
};
```

Слабостью этой структуры является то, что она не обладает гибкостью при ее изменении. Как только клиенты начнут использовать эту структуру, будет очень тяжело изменить ее. Придется изменять слишком много клиентского кода. Если бы мы позднее решили, что хотели бы вычислять **x** и **y** вместо того, чтобы хранить эти значения, мы были бы обречены на неудачу. У нас возникли бы аналогичные проблемы при запоздалом озарении, что программа должна хранить **x** и **y** в базе данных. Это реальная проблема при недостаточной инкапсуляции: имеется препятствие для будущих изменений реализации. Неинкапсулированное программное обеспечение негибко, и, в результате, оно не очень устойчиво. При изменении внешних условий программное обеспечение неспособно элегантно измениться вместе с ними. Не забывайте, что мы говорим здесь о практической стороне, а не о том, что является потенциально возможным. Понятно, что можно изменить структуру **Point**. Но,

если большой объем кода зависит от этой структуры, то такие изменения не являются практичными.

Перейдем к рассмотрению класса с интерфейсом, предлагающим клиентам возможности, подобные тем, которые предоставляет выше описанная структура, но с инкапсулированной реализацией:

```
class Point {
public:
    int getXValue() const;
    int getYValue() const;
    void setXValue(int newXValue);
    void setYValue(int newYValue);

private:
   ...      // прочее...
};
```

Этот интерфейс поддерживает реализацию, используемую структурой (сохраняющей **x** и **y** как целые), но он также предоставляет альтернативные реализации, основанные, например, на вычислении или просмотре базы данных. Это более гибкий замысел, и гибкость делает возникающее в результате программное обеспечение более устойчивым. Если реализация класса найдена недостаточной, она может быть изменена без изменения клиентского кода. Принятые объявления доступных методов остаются, неизменными, что ведет к неизменности клиентского исходного текста.

Инкапсулированное программное обеспечение более гибко, чем неинкапсулированное, и, при прочих равных условиях, эта гибкость делает его предпочтительнее при выборе метода проектирования.

Степень инкапсуляции

Класс, рассмотренный выше, не полностью инкапсулирует свою реализацию. Если реализация изменяется, то еще имеется код, который может быть изменен. В частности, методы класса могут оказаться нарушенными. По всей видимости, они зависят от особенностей данных класса. Однако ясно видно, что класс более инкапсулирован, чем структура, и хотелось бы иметь способ установить это более формально.

Это легко сделать. Причина, по которой класс является более инкапсулированным, чем структура, заключается в том, что при изменении открытых данных структуры может оказаться разрушенным больше кода, чем при изменением закрытых данных класса. Это ведет к следующему подходу в оценке двух реализаций инкапсуляции: если изменение для одной реализации может привести к большему разрушению кода, чем это разрушение будет при другой реализации, то соответствующее изменение для первой реализации, будет менее инкапсулировано. Это определение совместимо с нашей интуицией, которая подсказывает нам, что вносить изменения следует таким образом, чтобы разрушать как можно меньше кода. Имеется прямая связь между инкапсуляцией (сколько кода могут разрушить вносимые изменения) и практической гибкостью (вероятность, что мы будем делать специфические изменения).

Простой способ измерить, сколько кода может быть разрушено, состоит в том, чтобы считать функции, на которые пришлось бы воздействовать. То есть, если изменение одной реализации ведет потенциально к большему числу разрушаемых функций, чем изменения в другой реализации, то первая реализация менее инкапсулирована, чем вторая. Если мы применим эти рассуждения к описанной выше структуре, то увидим, что изменение ее элементов может разрушить неопределенно большое количество функций, а именно: каждую функцию, использующую эту структуру. В общем случае мы не можем рассчитать количество таких функций, потому что не имеется никакого способа выявить весь код, который использует специфику структуры. Это особенно видно, если изменения касаются кода библиотек. Однако число функций, которые могли бы быть разрушены, если изменить данные, являющиеся элементами класса, подсчитать просто: это все функции, которые имеют доступ к закрытой части класса. В данном случае, изменятся только четыре функции (не включая объявлений в закрытой части класса). И мы знаем об этом, потому что все они удобно перечислены при определении класса. Так как они — единственные функции, которые имеют доступ к закрытым частям класса, они также — единственные функции, на которые можно воздействовать, если эти части изменяются.

Инкапсуляция и функции — не члены

Приемлемый способом оценки инкапсуляции является количество функций, которые могли бы быть разрушены, если изменяется реализация класса. В этом случае становится ясно, что класс с **n** методами более инкапсулирован, чем класс с **n+1** методами. И это наблюдение поясняет предпочтение в выборе функций, не являющихся ни друзьями, ни методами: если функция **f** может быть выполнена как метод или как функция, не являющаяся другом, то создание ее в виде метода уменьшило бы инкапсуляцию, тогда как создание ее в виде «недруга» инкапсуляцию не уменьшит. Так как функциональность здесь не обсуждается (функциональные возможности **f** доступны классам клиентов независимо от того, где эта **f** размещена), мы естественно предпочитаем более инкапсулированный проект.

Важно, что мы пытаемся выбрать между методами класса и внешними функциями, не являющимися друзьями. Точно так же, как и методы, функции-друзья могут быть подвержены разрушениям при изменении реализации класса. Поэтому, выбор между методами и функциями-друзьями можно правильно сделать только на основе анализа поведения. Кроме того, общее мнение о том, что «функции-друзья нарушают инкапсуляцию» — не совсем истина. Друзья не нарушают инкапсуляцию, они только уменьшают ее точно таким же способом, что и методы класса.

Этот анализ применяется к любому виду методов, включая и статические. Добавление статического метода к классу, когда его функциональные возможности могут быть реализованы как не члены и не друзья уменьшают инкапсуляцию точно так же, как это делает добавление нестатического метода. Перемещение свободной функции в класс с оформлением ее в виде статического метода, только для того, чтобы показать, что она соприкасается с этим классом, является плохой идеей. Например, если имеется абстрактный класс для виджетов (Widgets) и затем используется функция фабрики классов, чтобы дать возможность клиентам создавать виджеты, можно использовать следующий общий, но худший способ организовать это:

```
// a design less encapsulated than it could be
class Widget {
   ...  // внутренее наполнение Widget; может быть:
        // public, private, или protected
```

```
public:
    // может быть также «недругом» и не членом
    static Widget* make(/* params */);
};
```

Лучшей идеей является создание вне Widget, что увеличивает совокупную инкапсуляцию системы. Чтобы показать, что Widget и его создание (make) все-таки связаны, используется соответствующее пространство имен (namespace):

```
// более инкапсулированный проект
namespace WidgetStuff {
    class Widget { ... };
    Widget* make( /* params */ );
};
```

Увы, у этой идеи имеется своя слабость, когда используются шаблоны.

Синтаксические проблемы

Возможно что вы, как и многие люди, имеете представление относительно синтаксического смысла утверждения, что не методы и не друзья предпочтительнее методов. Возможно, что вы даже «купились» на аргументы относительно инкапсуляции. Теперь, предположим, что класс **Wombat** поддерживает функциональные возможности *поедания* и *засыпания*. Далее предположим, что функциональные возможности, связанные с поеданием, должны быть выполнены как метод, а засыпание может быть выполнено как член или как не член и не друг. Если вы следуете советам, описанным выше, вы создадите описания подобные этим:

```
class Wombat {
public:
    void eat(double tonsToEat);
    ...
};
void sleep(Wombat& w, double hoursToSnooze);
```

Это привело бы к синтаксическому противоречию для клиентов класса, потому что для

```
Wombat w;
```

они напишут:

```
w.eat(.564);
```

при вызове **eat**. Но они написали бы:

```
sleep(w, 2.57);
```

для выполнения **sleep**. При использовании только методов класса можно было бы иметь более опрятный код:

```
class Wombat {
public:
   void eat(double tonsToEat);
   void sleep(double hoursToSnooze);
   ...
};
w.eat(.564);
w.sleep(2.57);
```

Ах, эта всеобщая однородность! Но эта однородность вводит в заблуждение, потому что в мире имеется огромное количество функций, которые мечтают о вашей философии.

Чтобы непосредственно ее использовать, нужны функции — не методы. Позвольте нам продолжить пример Wombat. Предположим, что вы пишете программу моделирования этих прожорливых созданий, и воображаете, что одним из методов, в котором вы часто нуждаетесь при использовании вомбатов, является засыпание на полчаса. Ясно, что вы можете засорить ваш код обращениями **w.sleep (.5)**, но этих (**.5**) будет так много, что их будет трудно напечатать. А что произойдет, если это волшебное значение должно будет измениться? Имеется ряд способов решить эту проблему, но возможно самый простой заключается в определении функции, которая инкапсулирует детали того, что вы хотите сделать. Понятно, что если вы не являетесь автором **Wombat**, функция будет обязательно внешней, и вы будете вызвать ее таким образом:

```
// может быть inline, но это не меняет сути
void nap(Wombat& w) { w.sleep(.5); }
Wombat w;
...
nap(w);
```

И там, где вы используете это, появится синтаксическая несогласованность, которой вы так боитесь. Когда вы хотите кормить ваши желудки (wombats), вы обращаетесь к методу

класса, но когда вы хотите их усыпить, вы обращаетесь к внешней функции.

Если вы самокритичны и честны сами с собой, вы увидите, что имеете эту предполагаемую несогласованность со всеми нетривиальными классами, вы используете ее потому, что класс не может имеет любую функцию, которую пожелает какой-то клиент. Каждый клиент добавляет, по крайней мере, несколько своих функций для собственного удобства, и эти функции всегда не являются методами классов. Программисты на C++ используют это, и они не думают ничего об этом. Некоторые вызовы используют синтаксис методов, а некоторые синтаксис внешних вызовов. Клиенты только ищут, какой из синтаксисов является соответствующим для тех функций, которые они хотят вызвать, затем они вызывают их. Жизнь продолжается. Это происходит особенно часто в STL (Стандартной библиотеки C++), где некоторые алгоритмы являются методами (например, **size**), некоторые — не методами (например, **unique**), а некоторые — тем и другим (например, **find**). Никто и глазом не моргает.

Интерфейсы и упаковка

«Интерфейс» класса (подразумевая, функциональные возможности, обеспечиваемые классом) включает также внешние функции, связанные с классом. Им также показано, что правила области видимости имен в C++ поддерживают эти изменения понятия «интерфейса». Это означает, что решение сделать функцию, зависимую от класса, в виде не друга — не члена вместо члена даже не изменяет интерфейс этого класса! Кроме того, вывод функций интерфейса класса за границы определения класса ведет к некоторой замечательной гибкости упаковки, которая была бы иначе недоступна. В частности, это означает, что интерфейс класса может быть разбит на множество заголовочных файлов.

Предположим, что автор класса **Wombat** обнаружил, что клиенты **Wombat** часто нуждаются в ряде дополнительных функций, связанных, с едой, сном и размножением. Такие функции по определению не строго необходимы. Те же самые функциональные возможности могли бы быть получены через вызов других (хотя и более громоздких) методов. В результате, каждая дополнительная функция должна быть не другом и не

методом. Но предположим, что клиенты дополнительных функций, используемых для еды, редко нуждаются в дополнительных функциях для сна или размножения. И предположим, что клиенту, использующему дополнительные функции для сна и размножения, также редко нужны дополнительные функции для еды. То же самое можно развить на функции размножения и сна.

Вместо размещения всех Wombat-зависимых функций в одном заголовочном файле, предпочтительнее было бы разместить элементы интерфейса **Wombat** в четырех отдельных заголовках: один для функций ядра **Wombat** (описания функций, связанных с определением класса), и по одному для каждой дополнительной функции, определяющей еду, сон, и размножение. Клиенты включают в свои программы только те заголовки, в которых они нуждаются. Возникающее в результате программное обеспечение не только более ясное, оно также содержит меньшее количество зависимостей, пустых для трансляции. Этот подход, использующий множество заголовков, был принят для стандартной библиотеки (STL). Содержание **namespace std** размещено в 50 различных заголовочных файлах. Клиенты включают заголовки, объявляющие только части библиотеки, необходимые им, и они игнорирует все остальное.

Кроме этого, такой подход расширяем. Когда объявления функций, составляющих интерфейс класса, распределены по многим заголовочным файлам, это становится естественным для клиентов, которые размещают свои наборы дополнительных функций, специфические для приложения, в новых заголовочных файлах, подключаемых дополнительно. Другими словами, чтобы обработать специфические для приложения дополнительные функции, они поступают точно так же, как и авторы класса. Это то, что и должно быть. В конце концов, это только дополнительные функции.

Минимальность и инкапсуляция

Существуют интерфейсы класса, которые являются полными и минимальными. Такие интерфейсы позволяют клиентам класса делать что-либо, что они могли бы предположительно хотеть делать, но классы содержат методов не больше, чем абсолютно необходимо. Добавление функций вне минимума необходимого для того, чтобы клиент мог сделать его

работу, уменьшает возможности повторного использования класса. Кроме того, увеличивается время трансляции для программы клиента. Добавление методов сверх этих требований нарушает принцип открытости-закрытости, производит жирные интерфейсы класса, и в конечном счете ведет к загниванию программного обеспечения. Возможно увеличение числа параметров, чтобы уменьшить число методов в классе, но теперь мы имеем дополнительную причину, чтобы отказаться от этого: уменьшается инкапсуляция класса.

Конечно, минимальный интерфейс класса — не обязательно самый лучший интерфейс. Добавление функций сверх необходимости может быть оправданным, если это значительно увеличивает эффективность класса, делает класс более легким в использовании или предотвращает вероятные клиентские ошибки. Основываясь на работе с различными строковыми классами, можно отметить, что для некоторых функций трудно ощутить, когда их делать не членами, даже если они могли быть не друзьями и не методами. «Наилучший» интерфейс для класса может быть найден только после балансировки между многими конкурирующими параметрами, среди которых степень инкапсуляции является лишь одним.

Стандартная мудрость, несмотря на использование «недругов» и не методов улучшает инкапсуляцию класса, и предпочтительнее для таких функций над методами, потому что делает решение проще, когда надо проектировать и разрабатывать классы с интерфейсами, которые являются полными и минимальными (или близкими к минимальному). Возражения, связанные с неестественностью, возникающей в результате изменения синтаксиса вызова, являются совершенно необоснованными, а склонность к «недругам» и не методам ведет к пакующим стратегиям для организации интерфейсов класса, которые минимизируют клиентские зависимости при трансляции, сохраняя доступ к максимальному числу дополнительных функций.

Пришло время отказаться от традиционной, но неточной идеи, связанной с тем, что означает «быть объектно-ориентированным». Действительно ли вы — истинный сторонник инкапсуляции? Если так, то вы ухватитесь за «недружественные» внешние функции с пылом, которого они заслуживают.

Все изложенное выше показывает, что не все так гладко в чистых методах объектно-ориентированного проектирования, если приходится прибегать к ухищрениям, присущим чисто процедурному программированию. Конечно, эффект от использования будет очевиден лишь при разработке достаточно больших программных систем, когда программу приходится развивать и модифицировать, а не создавать заново. Отсюда следует, что чисто объектные языки и методы могут оказаться в этом случае весьма неудобными. А значит: прощай Java и Си? Или их ждет ревизия?

С другой стороны оказывается, что инкапсуляция лучше всего удается процедурным языкам!? Ведь любую структуру данных можно окружить внешними функциями и через них осуществлять доступ. А методы в классе вообще не нужны!?

Обзор C/C++ компиляторов EMX и Watcom

Watcom C/C++

Watcom — звезда прошлого. Основные черты — многоплатформенность и качество кода. В лучшие времена генерировал код для DOS real mode, DOS protected mode (DOS/4G, DOS/4GW, Phar Lap), Win16, Win32, Win32s, QNX, OS/2 (16- и 32-bit), Netware NLM. Причем, работая под любой системой, можно было генерировать код для всех остальных (к примеру, программу под Win32 можно было скомпилировать и слинковать из-под OS/2 и т.д.). Watcom стал весьма популярен во времена DOS-игр, работающих в защищенном режиме (DOOM и прочие).

К моменту появления версии 11.0 (1997 г.) фирма, разрабатывавшая Watcom, была куплена Sybase Inc., и это, к сожалению, возвестило о кончине компилятора. Дальнейшая разработка была практически заморожена, а в 1999 г. Sybase Inc. объявила о прекращении продаж и установила крайний срок, после которого будет прекращена и техническая поддержка для тех, кто еще успел купить компилятор (это было в середине 2000 г.). Дальнейшая судьба продукта пока неизвестна.

Последняя версия — 11.0B. C++ компилятор в ней не поддерживает **namespaces** и не содержит STL. Впрочем,

существуют многие реализации STL, поддерживающие Watcom C++ (к примеру, STLPort).

Под любую поддерживаемую систему есть набор стандартных утилит: компиляторы, линкер, отладчик(и), **make**, **lib**, **strip** и другие. В системах с GUI (OS/2, Windows) есть также IDE (хотя и не очень удобная).

Кодогенерация застыла на уровне 1997 г., и теперь даже MS Visual C++ обгоняет Watcom (естественно, сравнения проводились под Windows, но некоторое представление это дать может).

При работе с Watcom C++ под OS/2 нужно знать следующее:

- В версиях 11.0* в линкере есть досадная ошибка, и вызовы 16-разрядных функций OS/2 (Vio*, Kbd*, Mou* и др.) будут давать трапы. Для борьбы с этим предназначена утилита **LXFix**, которая запускается после линкера и исправляет **fixups**.
- В комплект входит весьма древний OS/2 Toolkit (от OS/2 2.x). Поэтому крайне рекомендуется установить Toolkit из последних (4.0, 4.5).

Кроме разработки «родных» OS/2-программ, Watcom C/C++ можно рекомендовать для компиляции кода, слабо привязанного к ОС. Наличие в стандартной библиотеке функций вроде **_dos_setdrive()**, поддерживаемых под всеми системами (ну, или как минимум под OS/2, Win32 и DOS) позволяет писать в этом смысле платформонезависимо (для пользовательского интерфейса в данном случае можно использовать Turbo Vision).

И напоследок стоит еще раз напомнить про то, что компилятор более не развивается и не поддерживается. Имеющиеся проблемы никуда не денутся и не будут теперь решены.

EMX (GNU C/C++)

EMX — представительство Unix в OS/2 и одно из представительств Unix в DOS. Это целый комплект из компиляторов, сопутствующих утилит и библиотек поддержки. В первую очередь предназначен для портирования программ из среды Unix в OS/2, для чего эмулирует множество функций в

«первозданном» виде, включая даже и **fork()**. Основывается на одном из наибольших достижений мира бесплатных программ — системе компиляторов GCC (gcc означает «GNU Compiler Collection»). GCC состоит из собственно трансляторов с языков программирования (в настоящее время это C, C++, Objective C, Fortran 77, Chill и Java, хотя ничто не мешает встроить в систему свой язык), превращающих исходный код в программу на внутреннем языке компилятора (он называется RTL — Register Transfer Language) и стартующих уже от представления на RTL генераторов машинного кода для различных платформ. В частности, поддерживается платформа i386.

Сам EMX является портом GCC под OS/2/DOS и содержит измененные версии компиляторов, линкера, отладчика **gdb** и многих других программ; стандартную библиотеку C, содержащую множество функций из мира Unix; DLL поддержки и многое другое. Кроме того, с помощью EMX под OS/2 были скомпилированы многие другие Unix-программы, к примеру GNU Make, который обязательно понадобится при мало-мальски серьезной разработке.

Кроме всего прочего, EMX позволяет создавать «родные» программы для OS/2, используя OS/2 API. Можно также использовать в программах одновременно и «родные», и «заимствованные» функции.

Программы же, не использующие OS/2 API и некоторых функций Unix, будут «контрабандой» работать и из-под голого DOS во flat mode (в комплекте с EMX поставляется DOS-расширитель). К тому же, и под Windows есть расширитель **rsx.exe**, позволяющий запускать файлы в формате **a.out**, сгенерированные EMX!

Но сам GCC родом из мира Unix, и поэтому EMX также привносит с собой кое-что оттуда. Вот основные моменты:

- Прямой слэш ('/'). Как известно, в Unix для разделения каталогов в файловом пути вместо обратных слэшей используются прямые. Нет, все стандартные функции (**open()**, **fopen()** и др.) понимают оба варианта, но вот при указании файлов и путей компилятору придется использовать прямые. (Не пугайтесь, **c:/aaa/bbb/ccc —** это нормально.)

- Нестандартные форматы файлов. Да, объектные файлы имеют расширение **.o** и формат a**.out**, отличный от привычных **.obj**-файлов. То же самое верно и для файлов объектных библиотек (**.a** в сравнении с **.lib**). И даже исполняемые файлы фактически являются файлами формата **a.out**, содержащими пришпиленный в начале LX-загрузчик.

Но это еще не все. Существует возможность делать и **.obj** файлы, и нормальные **LX .exe** (для этого вызываются всяческие конверторы и на финальном этапе link386). Все эти многочисленные варианты (еще отметим широкие возможности по созданию различных типов DLL) разнятся предоставляемыми возможностями. К примеру, если работать с **.obj** и **LX .exe**, то программа не будет запускаться под DOS и ее нельзя будет отлаживать. Если к тому же выбрать статическую линковку, то еще и список поддерживаемых функций уменьшится. В общем, есть простор для экспериментирования (хотя наиболее часто используемый вариант — **a.out** формат исполняемого файла плюс динамическая линковка с EMX runtime).

- Расширения C, C++. Не будем их здесь перечислять, отметим лишь, что они есть и что их применение делает программу непереносимой.
- Компиляция всегда идет через ассемблер. Т.е. кодогенераторы генерируют лишь ассемблерный текст и переваливают проблему на плечи ассемблера. Не стоит пугаться, с ней он справляется весьма быстро. К тому же, существуют возможности:
    - всем частям компилятора общаться друг между другом с помощью pipes (выход препроцессора поступает сразу на вход компилятору, а выход последнего ассемблеру) и обойтись без временных файлов.
    - всем частям компилятора висеть некоторое время в памяти после последнего обращения и тем самым экономить время на их запуске.
- Нестандартная библиотека. Работавшие с какими-либо другими компиляторами могут не найти привычных функций, зато могут найти множество других, доселе неизвестных.

Но основное отличие EMX от остальных — это объединение хендлов файлов и сокетов в одну группу. К примеру, используя EMX, не нужно вызывать **sock_init()**, можно использовать **read()** и **write()**, а задачу **soclose()** выполняет обычный **close()**. Кроме этого, функция **select()**, работающая в IBM TCP/IP только для сокетов, в EMX расширена до поддержки любых хендлов, как и полагается в Unix'е.

Как уже отмечалось выше, GCC распространяется под лицензией GNU. Разработка GCC, инициированная где-то в конце 80-х гг. — начале 90-х гг., поначалу велась командой разработчиков, возглавлявшейся идеологом GNU Ричардом М. Столлменом (rms); в 1996 г. ими была выпущена версия 2.7.2.1 и затем экспериментальная версия 2.8.1. Если поддержка C в последней была на уровне ANSI C + расширения, то ситуация с C++ была тяжелой; к тому же, разработка фактически остановилась. Но еще до выпуска 2.8.1 за развитие GCC взялась фирма Cygnus, особенно направив свои усилия на выправление ситуации с C++ (к тому времени до принятия стандарта C++ оставалось не так уж и много). Эта фирма выпустила несколько версий EGCS (Enhanced GNU Compiler Suite), после чего Столлмен и компания решили и вовсе их благословить. Развитие версии 2.8.1, содержавшей кучу ошибок в реализации C++, было заброшено, последняя к тому времени версия EGCS автоматически превратилась в последнюю версию GCC (2.95), а развитие GCC фактически продолжилось командой из Cygnus. Последняя выпущенная ими версия — 2.95.2, это случилось 27 октября 1999 г. (А сама Cygnus не так давно была приобретена небезызвестной компанией Red Hat Inc.)

Последняя версия GCC довольно близка к стандарту, поддерживает все последние добавления к C++ (вроде namespace) и включает в себя также реализацию STL от SGI (она включена в libstdc++, последняя версия 2.90.8). STL из libstdc++ близка к стандарту, но iostreams там все еще не template-based, а взяты из совсем старой libg++. Впрочем, можно опять же обратиться к STLport, она поддерживает и GCC.

Таково состояние GCC на сегодняшний момент. Однако, использовать GCC под OS/2 означает использовать EMX, последняя версия которого (v0.9d) включает в себя старый GCC 2.8.1. Но все не так плохо. Ибо есть еще проект под названием PGCC, суть Pentium-optimized GCC. Сам GCC хоть и

содержит различные оптимизации для базовой платформы, но про особенности конкретных процессоров современности (а это кроме различных вариантов Pentium еще и Cyrix, AMD, все сильно отличающиеся друг от друга по тому, как надо для них оптимизировать) знает крайне мало. Цель проекта PGCC — научить GCC генерировать программы, выжимающие максимум из процессора. (PGCC — это набор «патчей» к GCC). Последний PGCC — 2.95.3, основан на GCC 2.95.2. Оптимизация для конкретного процессора производится при указании определенного ключа в командной строке, так что если его не указывать, то мы получаем «честный» GCC 2.95.2, со всеми его прелестями.

А теперь о прелестях применительно к OS/2. Сам компилятор версии 2.95.2 уже вполне неплох. Он параноидален в духе последнего стандарта (предупреждений об ошибках в сравнении с версией EGCS 1.1.2 стало раза в два больше), не падает, генерирует приемлемый код. Смелые могут даже поставить ключ **-O6** и попробовать оптимизацию под Pentium (здесь имеется в виду PGCC). Но про нормальную отладку PM-приложений можно сразу же забыть. Нацеленный на это PMGDB, входящий в состав EMX, крайне примитивен, да и порой просто не работает. То же самое с profiling (поддержка заявлена, но виснет намертво, до reset). Проблемы могут явиться сами собой. Короче говоря, будьте готовы к возникновению странных проблем и к дубовой отладке.

Компиляторы GCC (C и C++), как уже говорилось выше, можно рекомендовать для переноса программ из Unix под OS/2. Впрочем, как раз в этой области весьма мало вариантов, если не сказать, что как раз один. Можно наоборот, с помощью EMX разрабатывать программы, которые потом будут работать под Unix. Правда, к сожалению, многие функции не поддерживаются EMX. Как минимум, нет очередей сообщений, семафоров, shared memory (ни BSD, ни POSIX). Здесь стоит также заметить, что порты GCC существуют и под win32, и под DOS (а еще вспомним про возможность запуска a.out-программ, сделанных EMX, под DOS и win32!), так что теоретически с помощью EMX можно писать программы, которые будут компилироваться и работать под OS/2, Unix, DOS и Windows.

Главное же достоинство EMX — он абсолютно бесплатен и доступен в исходных текстах. А если вы не верите, что он

работает, вот доказательство: такая большая вещь, как XFree86, компилируется с помощью EMX и работает под OS/2! Не говоря о многих других программах меньшего размера.

Использование директивы #import

Как осуществить на VC создание документа и написать туда пару слов?

Возникла следующая проблема — необходимо загрузить документ Excel или Word (вместе с программами — т.е. запускается Word и загружается в него документ) и запустить в нем функцию или макрос на VBA.

Имеется файл БД. Необходимо читать и писать (добавлять и изменять) в файл. Как это лучше сделать?

Как работать с OLE?

Подобные вопросы часто можно встретить в конференциях Fidonet, посвящённых программированию на Visual C++. Как правило, после некоторого обсуждения, фидошная общественность приходит к мнению, что лучшее решение — использование директивы **#import**.

Ниже мы попытаемся объяснить то, как работает эта директива и привести несколько примеров её использования. Надеемся, после этого вы тоже найдёте её полезной.

Директива **#import** введена в Visual C++, начиная с версии 5.0. Её основное назначение облегчить подключение и использование интерфейсов COM, описание которых реализовано в библиотеках типов.

Библиотека типов представляет собой файл или компонент внутри другого файла, который содержит информацию о типе и свойствах COM объектов. Эти объекты представляют собой, как правило, объекты OLE автоматизации. Программисты, которые пишут на Visual Basic, используют такие объекты, зачастую сами того не замечая. Это связано с тем, что поддержка OLE автоматизации является неотъемлемой частью VB и при этом создаётся иллюзия того, что эти объекты также являются частью VB.

Добиться такого же эффекта при работе на C++ невозможно (да и нужно ли?), но можно упростить себе жизнь, используя

классы, представляющие обёртки (wrappers) интерфейса **IDispatch**. Таких классов в библиотеках VC имеется несколько.

Первый из них — **COleDispatchDriver**, входит в состав библиотеки MFC. Для него имеется поддержка со стороны MFC ClassWizard'a, диалоговое окно которого содержит кнопку **Add Class** и далее **From a type library**. После выбора библиотеки типов и указания интерфейсов, которые мы хотим использовать, будет сгенерирован набор классов, представляющих собой обёртки выбранных нами интерфейсов. К сожалению, **ClassWizard** не генерирует константы, перечисленные в библиотеке типов, игнорирует некоторые интерфейсы, добавляет к именам свойств префиксы **Put** и **Get** и не отслеживает ссылок на другие библиотеки типов.

Второй — **CComDispatchDriver** является частью библиотеки ATL. В VC нет средств, которые могли бы облегчить работу с этим классом, но у него есть одна особенность — с его помощью можно вызывать методы и свойства объекта не только по ID, но и по их именам, то есть использовать позднее связывание в полном объёме.

Третий набор классов — это результат работы директивы **#import**.

Последний способ доступа к объектам OLE Automation является наиболее предпочтительным, так как предоставляет достаточно полный и довольно удобный набор классов.

Рассмотрим пример.

Создадим IDL-файл, описывающий библиотеку типов. Наш пример будет содержать описание одного перечисляемого типа **SamplType** и описание одного объекта **ISamplObject**, который в свою очередь будет содержать одно свойство **Prop** и один метод **Method**.

```
import "oaidl.idl";
import "ocidl.idl";

[
    uuid(37A3AD11-F9CC-11D3-8D3C-0000E8D9FD76),
    version(1.0),
    helpstring("Sampl 1.0 Type Library")
]
```

```
library SAMPLLib
{
     importlib("stdole32.tlb");
     importlib("stdole2.tlb");

     typedef enum {
          SamplType1 = 1,
          SamplType2 = 2
     } SamplType;

     [
          object,
          uuid(37A3AD1D-F9CC-11D3-8D3C-0000E8D9FD76),
          dual,
          helpstring("ISamplObject Interface"),
          pointer_default(unique)
     ]
     interface ISamplObject : IDispatch
     {
          [propget, id(1)] HRESULT Prop([out, retval]
SamplType *pVal);
          [propput, id(1)] HRESULT Prop([in] SamplType
newVal);
          [id(2)] HRESULT Method([in] VARIANT Var,[in] BSTR
Str,[out, retval] ISamplObject** Obj);
     };

     [
          uuid(37A3AD1E-F9CC-11D3-8D3C-0000E8D9FD76),
          helpstring("SamplObject Class")
     ]
     coclass SamplObject
     {
          [default] interface ISamplObject;
     };
};
```

После подключения соответствующей библиотеки типов с помощью директивы **#import** будут созданы два файла, которые генерируются в выходном каталоге проекта. Это файл **sampl.tlh**, содержащий описание классов, и файл **sampl.tli**, который содержит реализацию членов классов. Эти файлы будут включены

в проект автоматически. Ниже приведено содержимое этих файлов.

```
#pragma once
#pragma pack(push, 8)

#include <comdef.h>

namespace SAMPLLib {

// Forward references and typedefs struct __declspec
(uuid("37a3ad1d-f9cc-11d3-8d3c-0000e8d9fd76"))
/* dual interface */ ISamplObject;
struct /* coclass */ SamplObject;

// Smart pointer typedef declarations _COM_SMARTPTR_TYPEDEF
(ISamplObject, __uuidof(ISamplObject));

// Type library items
enum SamplType
{
     SamplType1 = 1,
     SamplType2 = 2
};

struct __declspec(uuid("37a3ad1d-f9cc-11d3-8d3c-
0000e8d9fd76"))
ISamplObject : IDispatch
{
     // Property data
     __declspec(property(get=GetProp,put=PutProp)) enum
SamplType Prop;

     // Wrapper methods for error-handling
     enum SamplType GetProp ( );
     void PutProp (enum SamplType pVal );
     ISamplObjectPtr Method (const _variant_t & Var,_bstr_t
Str );

     // Raw methods provided by interface
     virtual HRESULT __stdcall get_Prop (enum SamplType *
```

```
pVal) = 0 ;
     virtual HRESULT __stdcall put_Prop (enum SamplType pVal)
= 0 ;
     virtual HRESULT __stdcall raw_Method (VARIANT Var,BSTR
Str,struct ISamplObject** Obj) = 0 ;
};

struct __declspec(uuid("37a3ad1e-f9cc-11d3-8d3c-
0000e8d9fd76")) SamplObject;

#include "debug\sampl.tli"

} // namespace SAMPLLib

#pragma pack(pop)
----------------------------------------------------------------
#pragma once

// interface ISamplObject wrapper method implementations

inline enum SamplType ISamplObject::GetProp ( ) {
     enum SamplType _result;
     HRESULT _hr = get_Prop(&_result);
     if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuid-
of(this));
     return _result;
}

inline void ISamplObject::PutProp ( enum SamplType pVal ) {
     HRESULT _hr = put_Prop(pVal);
     if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuid-
of(this));
}
inline ISamplObjectPtr ISamplObject::Method ( const _vari-
ant_t & Var, _bstr_t Str ) {
     struct ISamplObject * _result;
     HRESULT _hr = raw_Method(Var, Str, &_result);
     if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuid-
of(this));
     return ISamplObjectPtr(_result, false);
}
```

Первое на что следует обратить внимание — это на строчку файла **sampl.tlh**:

```
namespace SAMPLLib {
```

Это означает, что компилятор помещает описание классов в отдельное пространство имён, соответствующее имени библиотеки типов. Это является необходимым при использовании нескольких библиотек типов с одинаковыми именами классов, такими, например, как **IDocument**. При желании, имя пространства имён можно изменить или запретить его генерацию совсем:

```
#import "sampl.dll" rename_namespace("NewNameSAMPLLib")
#import "sampl.dll" no_namespace
```

Теперь рассмотрим объявление метода **Method**:

```
ISamplObjectPtr Method (const _variant_t & Var,_bstr_t Str);
```

Здесь мы видим использование компилятором классов поддержки COM. К таким классам относятся следующие.

- **_com_error**. Этот класс используется для обработки исключительных ситуаций, генерируемых библиотекой типов или каким либо другим классом поддержки (например, класс **_variant_t** будет генерировать это исключение, если не сможет произвести преобразование типов).
- **_com_ptr_t**. Этот класс определяет гибкий указатель для использования с интерфейсами COM и применяется при создании и уничтожении объектов.
- **_variant_t**. Инкапсулирует тип данных **VARIANT** и может значительно упростить код приложения, поскольку работа с данными **VARIANT** напрямую является несколько трудоёмкой.
- **_bstr_t**. Инкапсулирует тип данных **BSTR**. Этот класс обеспечивает встроенную обработку процедур распределения и освобождения ресурсов, а также других операций.

Нам осталось уточнить природу класса **ISamplObjectPtr**. Мы уже говорили о классе **_com_ptr_t**. Он используется для реализации smart-указателей на интерфейсы COM. Мы будем часто использовать этот класс, но не будем делать этого напрямую. Директива **#import** самостоятельно генерирует

определение smart-указателей. В нашем примере это сделано следующим образом.

```
// Smart pointer typedef declarations
_COM_SMARTPTR_TYPEDEF(ISamplObject,__uuidof(ISamplObject));
```

Это объявление эквивалентно следующему:

```
typedef _com_ptr_t<ISamplObject,&__uuidof(ISamplObject)>
ISamplObjectPtr
```

Использование smart-указателей позволяет не думать о счётчиках ссылок на объекты COM, т.к. методы **AddRef** и **Release** интерфейса **IUnknown** вызываются автоматически в перегруженных операторах класса **_com_ptr_t**.

Помимо прочих, этот класс имеет следующий перегруженный оператор:

```
Interface* operator->() const throw(_com_error);
```

где **Interface** — тип интерфейса, в нашем случае — это **ISamplObject**. Таким образом, мы сможем обращаться к свойствам и методам нашего COM объекта. Вот как будет выглядеть пример использования директивы **#import** для нашего примера:

```
#import "sampl.dll"

void SamplFunc ()
{
     SAMPLLib::ISamplObjectPtr obj;
     obj.CreateInstance(L"SAMPLLib.SamplObject");

SAMPLLib::ISamplObjectPtr obj2 = obj->Method(1l,L"12345");
     obj->Prop = SAMPLLib::SamplType2;
     obj2->Prop = obj->Prop;
}
```

Как видно из примера создавать объекты COM с использованием классов, сгенерированных директивой **#import**, достаточно просто. Во-первых, необходимо объявить smart-указатель на тип создаваемого объекта. После этого для создания экземпляра нужно вызвать метод **CreateInstance** класса **_com_ptr_t**, как показано в следующих примерах:

```
SAMPLLib::ISamplObjectPtr obj;
   obj.CreateInstance(L"SAMPLLib.SamplObject");
```

или

```
obj.CreateInstance(__uuidof(SamplObject));
```

Можно упростить этот процесс, передавая идентификатор класса в конструктор указателя:

```
SAMPLLib::ISamplObjectPtr obj(L"SAMPLLib.SamplObject");
```

или

```
SAMPLLib::ISamplObjectPtr obj(__uuidof(SamplObject));
```

Прежде чем перейти к примерам, нам необходимо рассмотреть обработку исключительных ситуаций. Как говорилось ранее, директива **#import** использует для генерации исключительных ситуаций класс **_com_error**. Этот класс инкапсулирует генерируемые значения **HRESULT**, а также поддерживает работу с интерфейсом **IErrorInfo** для получения более подробной информации об ошибке. Внесём соответствующие изменения в наш пример:

```
#import "sampl.dll"

void SamplFunc ()
{
     try {
          using namespace SAMPLLib;
          ISamplObjectPtr obj(L"SAMPLLib.SamplObject");
          ISamplObjectPtr obj2 = obj->Metod(1l,L"12345");
          obj->Prop = SAMPLLib::SamplType2;
          obj2->Prop = obj->Prop;
     } catch (_com_error& er) {
          printf("_com_error:\n"
    "Error        : %08lX\n"
    "ErrorMessage: %s\n"
    "Description : %s\n"
    "Source       : %s\n",
    er.Error(),
    (LPCTSTR)_bstr_t(er.ErrorMessage()),
    (LPCTSTR)_bstr_t(er.Description()),
    (LPCTSTR)_bstr_t(er.Source()));
     }
}
```

При изучении файла **sampl.tli** хорошо видно как директива **#import** генерирует исключения. Это происходит всегда при выполнении следующего условия:

```
if (FAILED(_hr)) _com_issue_errorex(_hr, this,
__uuidof(this));
```

Этот способ, безусловно, является универсальным, но могут возникнуть некоторые неудобства. Например, метод **MoveNext** объекта **Recordset ADO** возвращает код, который не является ошибкой, а лишь индицирует о достижении конца набора записей. Тем не менее, мы получим исключение. В подобных случаях придётся использовать либо вложенные операторы **try {} catch**, либо корректировать **wrapper**, внося обработку исключений непосредственно в тело сгенерированных процедур. В последнем случае, правда, придется подключать файлы ***.tlh** уже обычным способом, через **#include**. Но делать это никто не запрещает.

Наконец, настало время рассмотреть несколько практических примеров. Приведем четыре примера работы с MS Word, MS Excel, ADO DB и ActiveX Control. Первые три примера будут обычными консольными программами, в последнем примере покажем, как можно заменить класс **COleDispatchDriver** сгенерированный MFC Class Wizard'ом на классы полученные директивой **#import**.

Для первых двух примеров нам понадобится файл следующего содержания:

```
// Office.h
#define Uses_MSO2000
#ifdef Uses_MSO2000
// for MS Office 2000
#import "C:\Program Files\Microsoft Office\Office\MSO9.DLL"
#import "C:\Program Files\Common Files\Microsoft
Shared\VBA\VBA6\VBE6EXT.OLB"
#import "C:\Program Files\Microsoft
Office\Office\MSWORD9.OLB" \
         rename("ExitWindows","_ExitWindows")
#import "C:\Program Files\Microsoft Office\Office\EXCEL9.OLB"
\
         rename("DialogBox","_DialogBox") \
         rename("RGB","_RGB") \
         exclude("IFont","IPicture")
#import "C:\Program Files\Common Files\Microsoft
Shared\DAO\DAO360.DLL" \
         rename("EOF","EndOfFile") rename("BOF","BegOfFile")
#import "C:\Program Files\Microsoft Office\Office\MSACC9.OLB"
```

```
#else
// for MS Office 97
#import "C:\Program Files\Microsoft Office\Office\MSO97.DLL"
#import "C:\Program Files\Common Files\Microsoft
Shared\VBA\VBEEXT1.OLB"
#import "C:\Program Files\Microsoft
Office\Office\MSWORD8.OLB" \
         rename("ExitWindows","_ExitWindows")
#import "C:\Program Files\Microsoft Office\Office\EXCEL8.OLB"
\
         rename("DialogBox","_DialogBox") \
         rename("RGB","_RGB") \
         exclude("IFont","IPicture")
#import "C:\Program Files\Common Files\Microsoft
Shared\DAO\DAO350.DLL" \
         rename("EOF","EndOfFile")
         rename("BOF","BegOfFile")
#import "C:\Program Files\Microsoft Office\Office\MSACC8.OLB"
#endif
```

Этот файл содержит подключение библиотек типов MS Word, MS Excel и MS Access. По умолчанию подключаются библиотеки для MS Office 2000, если на вашем компьютере установлен MS Office 97, то следует закомментировать строчку:

```
#define Uses_MSO2000
```

Если MS Office установлен в каталог, отличный от «**C:\Program Files\Microsoft Office\Office**», то пути к библиотекам также следует подкорректировать. Обратите внимание на атрибут **rename**, его необходимо использовать, когда возникают конфликты имён свойств и методов библиотеки типов с препроцессором. Например, функция **ExitWindows** объявлена в файле **winuser.h** как макрос:

```
#define ExitWindows(dwReserved,Code)
ExitWindowsEx(EWX_LOGOFF,0xFFFFFFFF)
```

В результате, там, где препроцессор встретит имя **ExitWindows**, он будет пытаться подставлять определение макроса. Этого можно избежать при использовании атрибута **rename**, заменив такое имя на любое другое.

MS Word

```
// console.cpp : Defines the entry point for the console
application.

#include "stdafx.h"
#include <stdio.h>
#include "Office.h"

void main()
{
  ::CoInitialize(NULL);
  try {
    using namespace Word;
    _ApplicationPtr word(L"Word.Application");
    word->Visible = true;
    word->Activate();

    // создаём новый документ
    _DocumentPtr wdoc1 = word->Documents->Add();

    // пишем пару слов
    RangePtr range = wdoc1->Content;
    range->LanguageID = wdRussian;
    range->InsertAfter("Пара слов");

    // сохраняем как HTML
    wdoc1->SaveAs(&_variant_t("C:\\MyDoc\\test.htm"),
      &_variant_t(long(wdFormatHTML)));
// иногда придется прибегать к явному преобразованию типов,
// т.к. оператор преобразования char* в VARIANT* не
определён

    // открывает документ test.doc
    _DocumentPtr wdoc2 = word->Documents->Open
(&_variant_t("C:\\MyDoc\\test.doc"));
    // вызываем макрос
    word->Run("Macro1");

  } catch (_com_error& er) {
    char buf[1024];
    sprintf(buf,"_com_error:\n"
```

```
      "Error        : %08lX\n"
      "ErrorMessage: %s\n"
      "Description : %s\n"
      "Source       : %s\n",
      er.Error(),
      (LPCTSTR)_bstr_t(er.ErrorMessage()),
      (LPCTSTR)_bstr_t(er.Description()),
      (LPCTSTR)_bstr_t(er.Source()));

      CharToOem(buf,buf); // только для консольных приложений
      printf(buf);
    }
    ::CoUninitialize();
}
```

MS Excel

```
// console.cpp : Defines the entry point for the console
application.

#include "stdafx.h"
#include <stdio.h>
#include "Office.h"

void main()
{
    ::CoInitialize(NULL);
    try {
      using namespace Excel;
      _ApplicationPtr excel("Excel.Application");
      excel->Visible[0] = true;

      // создаём новую книгу
      _WorkbookPtr  book  = excel->Workbooks->Add();
      // получаем первый лист (в VBA нумерация с единицы)
      _WorksheetPtr sheet = book->Worksheets->Item[1L];
// Аналогичная конструкция на VBA выглядит так:
// book.Worksheets[1]
// В библиотеке типов Item объявляется как метод или
// свойство по умолчанию (id[0]), поэтому в VB его
// можно опускать. На C++ такое, естественно, не пройдёт.
      // заполняем ячейки
```

```
      sheet->Range["B2"]->FormulaR1C1 = "Строка 1";
      sheet->Range["C2"]->FormulaR1C1 = 12345L;
      sheet->Range["B3"]->FormulaR1C1 = "Строка 2";
      sheet->Range["C3"]->FormulaR1C1 = 54321L;
      // заполняем и активизируем итоговую строку
      sheet->Range["B4"]->FormulaR1C1 = "Итого:";
      sheet->Range["C4"]->FormulaR1C1 = "=SUM(R[-2]C:R[-1]C)";
      sheet->Range["C4"]->Activate();
      // делаем красиво
      sheet->Range["A4:D4"]->Font->ColorIndex = 27L;
      sheet->Range["A4:D4"]->Interior->ColorIndex = 5L;
// Постфикс L говорит, что константа является числом типа
// long.
// Вы всегда должны приводить числа к типу long или short
// при преобразовании их к _variant_t, т.к. преобразование
// типа int к _variant_t не реализовано. Это вызвано не
// желанием   разработчиков компилятора усложнить нам жизнь,
// а спецификой самого типа int.

   } catch (_com_error& er) {
      char buf[1024];
      sprintf(buf,"_com_error:\n"
      "Error        : %08lX\n"
      "ErrorMessage: %s\n"
      "Description : %s\n"
      "Source       : %s\n",
      er.Error(),
      (LPCTSTR)_bstr_t(er.ErrorMessage()),
      (LPCTSTR)_bstr_t(er.Description()),
      (LPCTSTR)_bstr_t(er.Source()));


      CharToOem(buf,buf); // только для консольных приложений
      printf(buf);
   }
   ::CoUninitialize();
}
```

ADO DB

```
// console.cpp : Defines the entry point for the console
application.
```

```
#include "stdafx.h"
#include <stdio.h>

#import "C:\Program Files\Common
Files\System\ado\msado20.tlb" \
        rename("EOF","ADOEOF") rename("BOF","ADOBOF")
 // оператор rename необходим, т.к. EOF определён как макрос
 // в файле stdio.h
using namespace ADODB;

void main()
{
  ::CoInitialize(NULL);
  try {
    // открываем соединение с БД
    _ConnectionPtr con("ADODB.Connection");
    con->Open(L"Provider=Microsoft.Jet.OLEDB.3.51;"
  L"Data Source=Elections.mdb","","",0);

    // открываем таблицу
    _RecordsetPtr rset("ADODB.Recordset");
    rset->Open(L"ElectTbl",(IDispatch*)con,
   adOpenDynamic,adLockOptimistic,adCmdTable);

    FieldsPtr flds = rset->Fields;

    // добавляем
    rset->AddNew();
    flds->Item[L"Фамилия"] ->Value = L"Пупкин";
    flds->Item[L"Имя"] ->Value = L"Василий";
    flds->Item[L"Отчество"] ->Value = L"Карлович";
    flds->Item[L"Голосовал ли"] ->Value = false;
    flds->Item[L"За кого проголосовал"] ->Value = L"Против
всех";
    rset->Update();

    // подменяем
    flds->Item[L"Голосовал ли"] ->Value = true;
    flds->Item[L"За кого проголосовал"] ->Value = L"За
наших";
    rset->Update();
```

```
        // просмотр
        rset->MoveFirst();
        while (!rset->ADOEOF) {
          char buf[1024];
          sprintf(buf,"%s %s %s: %s - %s\n",
          (LPCTSTR)_bstr_t(flds->Item[L"Фамилия"]->Value),
          (LPCTSTR)_bstr_t(flds->Item[L"Имя"]->Value),
          (LPCTSTR)_bstr_t(flds->Item[L"Отчество"]->Value),
          (bool)flds->Item[L"Голосовал ли"]->Value? "Да": "Нет",
          (LPCTSTR)_bstr_t(flds->Item[L"За кого проголосовал"]
->Value));

          CharToOem(buf,buf);
          printf(buf);
          rset->MoveNext();
        }
      } catch (_com_error& er) {
        char buf[1024];
        sprintf(buf,"_com_error:\n"
        "Error        : %08lX\n"
        "ErrorMessage: %s\n"
        "Description : %s\n"
        "Source       : %s\n",
        er.Error(),
        (LPCTSTR)_bstr_t(er.ErrorMessage()),
        (LPCTSTR)_bstr_t(er.Description()),
        (LPCTSTR)_bstr_t(er.Source()));

        CharToOem(buf,buf); // только для консольных приложений
        printf(buf);
      }
      ::CoUninitialize();
    }
```

AciveX Control

Для этого примера нам понадобится любое оконное приложение.

ActiveX Control'ы вставляются в диалог обычно через **Components and Controls Gallery**: **Menu ➪ Project ➪ Add To Project ➪ Components and Controls-Registered ActiveX Controls**.

Нам в качестве примера вполне подойдёт **Microsoft FlexGrid Control**. Нажмите кнопку **Insert** для добавления его в проект, в появившемся окне **Confirm Classes** оставьте галочку только возле элемента **CMSFlexGrid** и смело жмите **OK**. В результате будут сформированы два файла **msflexgrid.h** и **msflexgrid.cpp**, большую часть содержимого которых нам придётся удалить. После всех изменений эти файлы будут иметь следующий вид:

msflexgrid.h

```
// msflexgrid.h

#ifndef __MSFLEXGRID_H__
#define __MSFLEXGRID_H__

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#pragma warning(disable:4146)
#import <MSFLXGRD.OCX>

class CMSFlexGrid : public CWnd
{
protected:
   DECLARE_DYNCREATE(CMSFlexGrid)
public:

  MSFlexGridLib::IMSFlexGridPtr I; // доступ к интерфейсу
  void PreSubclassWindow ();        // инициализация I
};

//{{AFX_INSERT_LOCATION}}

#endif
```

msflexgrid.cpp

```
// msflexgrid.cpp

#include "stdafx.h"
#include "msflexgrid.h"

IMPLEMENT_DYNCREATE(CMSFlexGrid, CWnd)
```

```
void CMSFlexGrid::PreSubclassWindow ()
{
  CWnd::PreSubclassWindow();

  MSFlexGridLib::IMSFlexGrid *pInterface = NULL;

  if (SUCCEEDED(GetControlUnknown()-
>QueryInterface(I.GetIID(),
     (void**)&pInterface))) {
    ASSERT(pInterface != NULL);
    I.Attach(pInterface);
  }
}
```

Теперь вставим элемент в любой диалог, например **CAboutDlg**. В диалог добавим переменную связанную с классом **CMSFlexGrid** и метод **OnInitDialog**, текст которого приведён ниже. При вызове диалога в наш **FlexGrid** будут добавлены два элемента:

```
BOOL CAboutDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  m_grid.I->AddItem("12345");
  m_grid.I->AddItem("54321");

  return TRUE;
}
```

В заключении, позволим ещё несколько замечаний.

Всегда внимательно изучайте файлы ***.tlh**. Отчасти они могут заменить документацию, а если её нет, то это единственный источник информации (кроме, конечно, OLE/COM Object Viewer).

Избегайте повторяющихся сложных конструкций. Например, можно написать так:

```
book->Worksheets->Item[1L]->Range["B2"]->FormulaR1C1 =
"Строка 1";
book->Worksheets->Item[1L]->Range["C2"]->FormulaR1C1 =
12345L;
```

Но в данном случае вы получите неоправданное замедление из-за лишнего межзадачного взаимодействия, а в случае DCOM — сетевого взаимодействия. Лучше написать так:

```
_WorksheetPtr sheet = book->Worksheets->Item[1L];
sheet->Range["B2"]->FormulaR1C1 = "Строка 1";
sheet->Range["C2"]->FormulaR1C1 = 12345;
```

При работе с MS Office максимально используйте возможности VBA для подготовки и тестирования вашего кода.

Будьте внимательны с версиями библиотек типов. К примеру, в MS Word 2000 появилась новая версия метода **Run**. Старая тоже осталась, но она имеет теперь название **RunOld**. Если вы используете MS Word 2000 и вызываете метод **Run**, то забудьте о совместимости с MS Word 97 — метода с таким ID в MS Word 97 просто нет. Используйте вызов **RunOld** и проблем не будет, хотя если очень хочется можно всегда проверить номер версии MS Word.

Бывают глюки. Сразу заметим, что это не связано с самой директивой **#import**. Например, при использовании класса **COleDispatchDriver** с MSADODC.OCX всё прекрасно работало, а после того как стали использовать директиву **#import**, свойство **ConnectionString** отказалось возвращать значение. Дело в том, что директива **#import** генерирует обёртку, используя dual-интерфейс объекта, а класс **COleDispatchDriver** вызывает **ConnectionString** через **IDispatch::Invoke**. Ошибка, видимо, в реализации самого MSADODC.OCX. После изменения кода вызова свойства всё заработало:

```
inline _bstr_t IAdodc::GetConnectionString () {
    BSTR _result;
    HRESULT _hr =
_com_dispatch_propget(this,0x01,VT_BSTR,&_result);
//  HRESULT _hr = get_ConnectionString(&_result);
   if (FAILED(_hr)) _com_issue_errorex(_hr, this,
__uuidof(this));
    return _bstr_t(_result, false);
}
```

В результате раскрутки библиотек типов MS Office, компилятор нагенерирует вам в выходной каталог проекта около 12 Mb исходников. Всё это он потом, естественно, будет компилировать. Если вы не являетесь счастливым обладателем

PIII, то наверняка заметите некоторые тормоза. В таких случаях надо стараться выносить в отдельный файл всю работу, связанную с подобными библиотеками типов. Кроме того, компилятор может генерировать обёртки классов каждый раз после внесения изменений в файл, в который включена директива **#import**. Представьте, что будет, если после каждого нажатия клавиши будут заново генерироваться все 12 Mb? Лучше вынести объявление директивы **#import** в отдельный файл и подключать его через **#include**.

Удачи в бою.

Создание системных ловушек Windows на Borland C++ Builder 5

Для начала определим, что именно мы хотим сделать.

Цель: написать программу, которая будет вызывающую хранитель экрана при перемещении курсора мыши в правый верхний угол и выдавать звуковой сигнал через встроенный динамик при переключении языка с клавиатуры.

Предполагается, что такая программа должна иметь небольшой размер. Поэтому будем писать её с использованием только WIN API.

Понятие ловушки

Ловушка (hook) — это механизм, который позволяет производить мониторинг сообщений системы и обрабатывать их до того как они достигнут целевой оконной процедуры.

Для обработки сообщений пишется специальная функция (**Hook Procedure**). Для начала срабатывания ловушки эту функцию следует специальным образом «подключить» к системе.

Если надо отслеживать сообщения всех потоков, а не только текущего, то ловушка должна быть глобальной. В этом случае функция ловушки должна находиться в DLL.

Таким образом, задача разбивается на две части:

- Написание DLL с функциями ловушки (их будет две: одна для клавиатуры, другая для мыши).
- Написание приложения, которое установит ловушку.

- Написание DLL.
- Создание пустой библиотеки.

C++ Builder имеет встроенный мастер по созданию DLL. Используем его, чтобы создать пустую библиотеку. Для этого надо выбрать пункт меню **File ➪ New**: В появившемся окне надо выбрать «**DLL Wizard**» и нажать кнопку «**OK**». В новом диалоге в разделе «**Source Type**» следует оставить значение по умолчанию — «**C++**». Во втором разделе надо снять все флажки. После нажатия кнопки «**Ок**» пустая библиотека будет создана.

Глобальные переменные и функция входа (DllEntryPoint)

Надо определить некоторые глобальные переменные, которые понадобятся в дальнейшем.

```
#define UP 1// Состояния клавиш
#define DOWN 2
#define RESET 3

int iAltKey; // Здесь хранится состояние клавиш
int iCtrlKey;
int iShiftKey;

int KEYBLAY;// Тип переключения языка
bool bSCRSAVEACTIVE;// Установлен ли ScreenSaver
MOUSEHOOKSTRUCT* psMouseHook;// Для анализа сообщений от
мыши
```

В функции **DllEntryPoint** надо написать код, подобный нижеприведённому:

```
if(reason==DLL_PROCESS_ATTACH)// Проецируем на адр. простр.
{
HKEY pOpenKey;
 char* cResult="";     // Узнаём как перекл. раскладка
long lSize=2;
KEYBLAY=3;

if(RegOpenKey(HKEY_USERS,".Default\\keyboard layout\\toggle",
&pOpenKey)==ERROR_SUCCESS)
{
   RegQueryValue(pOpenKey,"",cResult,&lSize);

if(strcmp(cResult,"1")==0)
```

```
        KEYBLAY=1;        // Alt+Shift
if(strcmp(cResult,"2")==0)
        KEYBLAY=2;        // Ctrl+Shift

RegCloseKey(pOpenKey);
}
 else
MessageBox(0,"Не могу получить данные о способе"
              "переключения раскладки клавиатуры",
         "Внимание!",MB_ICONERROR);
//------------- Есть ли активный хранитель экрана
if(!SystemParametersInfo(SPI_GETSCREENSAVEACTIVE,0,&bSCRSAVEAC
TIVE,0))
MessageBox(0,"Не могу получить данные об установленном"
              "хранителе экрана", "Внимание!",MB_ICONERROR);
 }
return 1;
```

Этот код позволяет узнать способ переключения языка и установить факт наличия активного хранителя экрана. Обратите внимание на то, что этот код выполняется только когда библиотека проецируется на адресное пространство процесса — проверяется условие (**reason==DLL_PROCESS_ATTACH**).

Функция ловушки клавиатуры

Функция ловушки в общем виде имеет следующий синтаксис:

```
LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM
lParam),
```

где:

- **HookProc** — имя функции;
- **nCode** — код ловушки, его конкретные значения определяются типом ловушки;
- **wParam, lParam** — параметры с информацией о сообщении.

В случае нашей задачи функция должна определять состояние клавиш **Alt**, **Ctrl** и **Shift** (нажаты или отпущены). Информация об этом берётся из параметров **wParam** и **lParam**. После определения состояния клавиш надо сравнить его со способом переключения языка (определяется в функции входа).

Если текущая комбинация клавиш способна переключить язык, то надо выдать звуковой сигнал.

Всё это реализует примерно такой код:

```
LRESULT CALLBACK KeyboardHook(int nCode,WPARAM wParam,LPARAM
lParam)
{         // Ловушка клав. - биканье при перекл. раскладки
if((lParam>>31)&1)           // Если клавиша нажата...
switch(wParam)
       {// Определяем какая именно
       case VK_SHIFT: {iShiftKey=UP; break};
       case VK_CONTROL: {iCtrlKey=UP; break};
       case VK_MENU: {iAltKey=UP; break};
}
else// Если была отпущена...
switch(wParam)
{// Определяем какая именно
       case VK_SHIFT: {iShiftKey=DOWN; break};
       case VK_CONTROL: {iCtrlKey=DOWN; break};
       case VK_MENU: {iAltKey=DOWN; break};
}
//--------------

switch(KEYBLAY)           // В зависимости от способа
переключения раскладки
{
    case 1: // Alt+Shift
{
if(iAltKey==DOWN && iShiftKey==UP)
    {
vfBeep();
      iShiftKey=RESET;
    }
if(iAltKey==UP && iShiftKey==DOWN)
    {
vfBeep();
      iAltKey=RESET;
    }
((iAltKey==UP && iShiftKey==RESET)||(iAltKey==RESET &&
iShiftKey==UP))
    {
```

```
      iAltKey=RESET;
      iShiftKey=RESET;
    }
       break;
}
//-----------------------------------
case 2: // Ctrl+Shift
       {
if(iCtrlKey==DOWN && iShiftKey==UP)
    {
vfBeep();
      iShiftKey=RESET;
    }
if(iCtrlKey==UP && iShiftKey==DOWN)
    {
vfBeep();
      iCtrlKey=RESET;
    }
if((iCtrlKey==UP && iShiftKey==RESET)||(iCtrlKey==RESET &&
iShiftKey==UP))
    {
      iCtrlKey=RESET;
      iShiftKey=RESET;
    }
}
}

return 0;
}
```

Звуковой сигнал выдаётся такой небольшой функцией:

```
void vfBeep()
{// Биканье
MessageBeep(-1);
MessageBeep(-1);// Два раза - для отчётливости
}
```

Функция ловушки мыши

Эта функция отслеживает движение курсора мыши, получает его координаты и сравнивает их с координатами правого верхнего угла экрана (0,0). Если эти координаты совпадают, то вызывается хранитель экрана. Для отслеживания движения

анализируется значение параметра **wParam**, а для отслеживания координат значение, находящееся в структуре типа **MOUSEHOOKSTRUCT**, на которую указывает **lParam**. Код, реализующий вышесказанное, примерно такой:

```
LRESULT CALLBACK MouseHook(int nCode,WPARAM wParam,LPARAM
lParam)
{        // Ловушка мыши - включает хранитель когда в углу
if(wParam==WM_MOUSEMOVE || wParam==WM_NCMOUSEMOVE)
{
 psMouseHook=(MOUSEHOOKSTRUCT*)(lParam);
if(psMouseHook->pt.x==0 && psMouseHook->pt.y==0)
if(bSCRSAVEACTIVE)
PostMessage(psMouseHook->hwnd,WM_SYSCOMMAND,
SC_SCREENSAVE,0);
}
return 0;
}
```

Обратите внимание, что команда на активизацию хранителя посылается в окно, получающее сообщения от мыши:

```
PostMessage(psMouseHook->hwnd,WM_SYSCOMMAND,
SC_SCREENSAVE ,0).
```

Теперь, когда функции ловушек написаны, надо сделать так, чтобы они были доступны из процессов, подключающих эту библиотеку. Для этого перед функцией входа следует добавить такой код:

```
extern "C" __declspec(dllexport) LRESULT CALLBACK
KeyboardHook(int,WPARAM,LPARAM);
extern "C" __declspec(dllexport) LRESULT CALLBACK
MouseHook(int,WPARAM,LPARAM);
```

Написание приложения, устанавливающего ловушку

Создание пустого приложения

Для создания пустого приложения воспользоваться встроенным мастером. Для этого надо использовать пункт меню **File ⇨ New**: В появившемся окне необходимо выбрать «**Console Wizard**» и нажать кнопку «**Ok**». В новом диалоге в разделе «**Source Type**» следует оставить значение по умолчанию — «**C++**». Во втором разделе надо снять все флажки. По нажатию «**Ок**» приложение создаётся.

Создание главного окна

Следующий этап — это создание главного окна приложения. Сначала надо зарегистрировать класс окна. После этого создать окно. Всё это делает следующий код (описатель окна **MainWnd** определён глобально):

```
BOOL InitApplication(HINSTANCE hinstance,int nCmdShow)
{ // Создание главного окна
WNDCLASS  wcx; // Класс окна
wcx.style=NULL;
wcx.lpfnWndProc=MainWndProc;
wcx.cbClsExtra=0;
wcx.cbWndExtra=0;
wcx.hInstance=hinstance;
wcx.hIcon=LoadIcon(hinstance,"MAINICON");
wcx.hCursor=LoadCursor(NULL,IDC_ARROW);
wcx.hbrBackground=(HBRUSH)(COLOR_APPWORKSPACE);
wcx.lpszMenuName=NULL;
wcx.lpszClassName="HookWndClass";

if(RegisterClass(&wcx))   // Регистрируем класс
{

MainWnd=CreateWindow("HookWndClass","SSHook", /* Создаём окно
*/
        WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT,CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT,
NULL,NULL,hinstance,NULL);
if(!MainWnd)
return FALSE;

return TRUE;
}
   return false;
}
```

Обратите внимание на то, каким образом был получен значок класса:

```
wcx.hIcon=LoadIcon(hinstance,"MAINICON");
```

Для того, чтобы это получилось надо включить в проект файл ресурсов (***.res**), в котором должен находиться значок с именем «**MAINICON**».

Это окно никогда не появится на экране, поэтому оно имеет размеры и координаты, устанавливаемые по умолчанию. Оконная процедура такого окна необычайно проста:

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT uMsg,WPARAM
wParam,
          LPARAM lParam)
{// Оконная процедура
switch (uMsg)
{
case WM_DESTROY:{PostQuitMessage(0); break;}
case MYWM_NOTIFY:
{
if(lParam==WM_RBUTTONUP)
PostQuitMessage(0);
break; // Правый щелчок на значке - завершаем
}
default:
return DefWindowProc(hwnd,uMsg,wParam,lParam);
}
return 0;
}
```

Размещение значка в системной области

Возникает естественный вопрос: если окно приложения никогда не появится на экране, то каким образом пользователь может управлять им (например, закрыть)? Для индикации работы приложения и для управления его работой поместим значок в системную область панели задач. Делается это следующей функцией:

```
void vfSetTrayIcon(HINSTANCE hInst)
{ // Значок в Tray
char* pszTip="Хранитель экрана и раскладка";
// Это просто Hint
NotIconD.cbSize=sizeof(NOTIFYICONDATA);
NotIconD.hWnd=MainWnd;
NotIconD.uID=IDC_MYICON;
NotIconD.uFlags=NIF_MESSAGE|NIF_ICON|NIF_TIP;
NotIconD.uCallbackMessage=MYWM_NOTIFY;
```

```
  NotIconD.hIcon=LoadIcon(hInst,"MAINICON");
lstrcpyn(NotIconD.szTip,pszTip,sizeof(NotIconD.szTip));
Shell_NotifyIcon(NIM_ADD,&NotIconD);
}
```

Для корректной работы функции предварительно нужно определить уникальный номер значка (параметр **NotIconD.uID**) и его сообщение (параметр **NotIconD.uCallbackMessage**). Делаем это в области определения глобальных переменных:

```
#define MYWM_NOTIFY (WM_APP+100)
#define IDC_MYICON  1006
```

Сообщение значка будет обрабатываться в оконной процедуре главного окна **(NotIconD.hWnd=MainWnd)**:

```
case MYWM_NOTIFY:
{
if(lParam==WM_RBUTTONUP)
PostQuitMessage(0);
break; // Правый щелчок на значке - завершаем
}
```

Этот код просто завершает работу приложения по щелчку правой кнопкой мыши на значке.

При завершении работы значок надо удалить:

```
void vfResetTrayIcon()
{// Удаляем значок
Shell_NotifyIcon(NIM_DELETE,&NotIconD);
}
```

Установка и снятие ловушек

Для получения доступа в функциям ловушки надо определить указатели на эти функции:

```
LRESULT CALLBACK (__stdcall *pKeybHook)(int,WPARAM,LPARAM);
LRESULT CALLBACK (__stdcall *pMouseHook)(int,WPARAM,LPARAM);
```

После этого спроецируем написанную DLL на адресное пространство процесса:

```
hLib=LoadLibrary("SSHook.dll");
(hLib описан как HINSTANCE hLib)
```

После этого мы должны получить доступ к функциям ловушек:

```
(void*)pKeybHook=GetProcAddress(hLib,"KeyboardHook");
(void*)pMouseHook=GetProcAddress(hLib,"MouseHook");
```

Теперь всё готово к постановке ловушек. Устанавливаются они с помощью функции **SetWindowsHookEx**:

```
hKeybHook=SetWindowsHookEx(WH_KEYBOARD,(HOOKPROC)(pKeybHook),h
Lib,0);
hMouseHook=SetWindowsHookEx(WH_MOUSE,(HOOKPROC)(pMouseHook),
hLib,0);
(hKeybHook  и hMouseHook описаны как HHOOK hKeybHook; HOOK
hMouseHook;)
```

Первый параметр — тип ловушки (в данном случае первая ловушка для клавиатуры, вторая — для мыши). Второй — адрес процедуры ловушки. Третий — описатель DLL-библиотеки. Последний параметр — идентификатор потока, для которого будет установлена ловушка. Если этот параметр равен нулю (как в нашем случае), то ловушка устанавливается для всех потоков.

После установки ловушек они начинают работать. При завершении работы приложения следует их снять и отключить DLL. Делается это так:

```
UnhookWindowsHookEx(hKeybHook);
UnhookWindowsHookEx(hMouseHook);   // Завершаем
FreeLibrary(hLib);
```

Функция WinMain

Последний этап — написание функции WinMain в которой будет создаваться главное окно, устанавливаться значок в системную область панели задач, ставиться и сниматься ловушки. Код её должен быть примерно такой:

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,LPSTR lpCmdLine,
   int nCmdShow)
{
MSG msg;
//----------------
hLib=LoadLibrary("SSHook.dll");
if(hLib)
{
(void*)pKeybHook=GetProcAddress(hLib,"KeyboardHook");
hKeybHook=SetWindowsHookEx(WH_KEYBOARD,(HOOKPROC)(pKeybHook),
   hLib,0);// Ставим ловушки
(void*)pMouseHook=GetProcAddress(hLib,"MouseHook");
hMouseHook=SetWindowsHookEx(WH_MOUSE,(HOOKPROC)(pMouseHook),
```

```
 hLib,0);
//-------------------------------
if (InitApplication(hInstance,nCmdShow))
// Если создали главное окно
{
vfSetTrayIcon(hInstance);// Установили значок
while (GetMessage(&msg,(HWND)(NULL),0,0))
{// Цикл обработки сообщений
TranslateMessage(&msg);
DispatchMessage(&msg);
}
//--------- Всё - финал
UnhookWindowsHookEx(hKeybHook); // Снимаем ловушки
UnhookWindowsHookEx(hMouseHook);
FreeLibrary(hLib);// Отключаем DLL
vfResetTrayIcon();// Удаляем значок
return 0;
}
}
return 1;
}
```

После написания этой функции можно смело запускать полностью готовое приложение.

Тонкости и хитрости в вопросах и ответах

Я наследовал из абстрактного класса A класс B и определил все pure-методы. А она при выполнении ругается, что в конструкторе A по прежнему зовётся абстрактный метод? Почему и что делать?

Так и должно быть — в C++ конструкторы предков вызываются только до конструктора потомка, а вызов методов не инициализированного потомка может окончиться катастрофически (это верно и для деструкторов).

Поэтому и была ограничена виртуальность при прямом или косвенном обращении в конструкторе (деструкторе) предка к виртуальным методам таким образом, что всегда будут вызваны методы предка, даже если они переопределены в потомке.

Замечание: это достижимо подменой **VMT**.

Практически принятое ограничение поначалу сбивает с толку, а проблемы с TV (созданным в Турбо Паскале, где конструктор сам зовёт нужные методы и конструкторы предков) доказывают незавершённость схемы конструкторов C++, в котором из-за автоматического вызова конструкторов подобъектов (предков) сложно описать конструирование объекта как целого в одном месте, поэтому конструкторы C++ правильнее называть **инициализаторами**.

Таким образом, логичнее было бы иметь два шага: автоматический вызов инициализаторов предков (например, с обнулением указателей) и последующий автоматический же вызов общего конструктора. И в C++ это реализуемо!

Для этого во всех классах нужно ввести инициализаторы (защищённые конструктор по умолчанию или конструкторы с фиктивным параметром) и в конструкторах потомка явно задавать именно их (чтобы подавить вызов конструкторов вместо инициализаторов предков). Если же код конструкторов выносить в отдельные (виртуальные) методы, то можно будет вызывать их в конструкторах потомков.

С деструкторами сложнее, поскольку в классе их не может быть более одного, поэтому можно ввести отдельные методы (типа **shutdown** и **destroy** в TV).

Теперь остаётся либо убрать деструкторы (хотя придётся явно вызывать методы деструкции), либо ввести общий признак, запрещающий деструкторам предков разрушать, и при переопределении метода деструкции переопределять также деструктор. И не забывайте делать их виртуальными!

В качестве примера можно привести преобразование следующего фрагмента, содержащего типичную ошибку, в правильный:

```
class PrintFile
public:
PrintFile(char name[]) Печать(GetFileName(name, MyExt()));
virtual const char *MyExt() return "xxx";
;
class PrintAnotherTypeOfFile :public PrintFile
public:
PrintAnotherTypeOfFile(char name[]) :PrintFile(name) const
char *MyExt() return "yyy";
;
```

После преобразования получаем следующее:

```
class PrintFile
enum Init_ Init; // Тип фиктивного параметра protected:
```

Инициализатор; здесь можно заменить на дефолт конструктор **PrintFile(Init_)**.

Можно добавить несколько «конструкторов» с другими именами, или, если «конструкторы» не виртуальные, можно использовать полиморфизм:

```
bool construct(char name[])
return Печать(GetFileName(name,MyExt()));
public:
//... Код вынесен в отдельный метод для использования в
потомках
PrintFile(char name[]) construct(name); virtual const char
*MyExt() return "xxx";
;
class PrintAnotherTypeOfFile :public PrintFile
//... Здесь инициализатор пропущен (никто не наследует)
```

```
public:
//... Конструктор; использует "конструктор" предка, с
виртуальностью;
//... указание инициализатора обязательно
PrintAnotherTypeOfFile(char name[]) :PrintFile(Init)
construct(name);

const char *MyExt() return "yyy";
;
```

Что такое NAN?

Специальное значение вещественного числа, обозначающее не-число — **Non-a-Number**. Имеет характеристику (смещенный порядок) из всех единиц, любой знак и любую мантиссу за исключением **.00__00** (такая мантисса обозначает бесконечность). Имеются даже два типа **не_чисел**:

- **SNAN** — Signalling NAN (сигнализирующие не-числа) — старший бит мантиссы=0
- **QNAN** — Quiet NAN (тихие не-числа) — старший бит мантиссы = 1.

SNAN никогда не формируется **FPU** как результат операции, но может служить аргументом команд, вызывая при этом случай недействительной операции.

QNAN=11__11.100__00 (называется еще «вещественной неопределенностью»), формируется **FPU** при выполнении недействительной операции, делении **0** на **0**, умножении **0** на **бесконечность**, извлечении корня **FSQRT**, вычислении логарифма **FYL2X** отрицательного числа, и т.д. при условии, что обработчик таких особых случаев замаскирован (регистр **CW**, бит **IM=1**). В противном случае вызывается обработчик прерывания **(Int 10h)** и операнды остаются неизменными.

Остальные не-числа могут определяться и использоваться программистом для облегчения отладки (например, обработчик может сохранить для последующего анализа состояние задачи в момент возникновения особого случая).

Как выключить оптимизацию и как longjmp может привести к баге без этого?

Иногда бывает необходимо проверить механизм генерации кода, скорость работы с какой-нибудь переменной или просто

использовать переменную в параллельных процедурах (например, обработчиках прерываний). Чтобы компилятор не изничтожал такую переменную и не делал её регистровой придумали ключевое слово **volatile**.

longjmp получает переменная типа **jmp_buf**, в которой **setjmp** сохраняет текущий контекст (все регистры), кроме значения переменных. То есть если между **setjmp** и **longjmp** переменная изменится, её значение восстановлено не будет.

Содержимое переменной типа **jmp_buf** никто никогда (кроме **setjmp**) не модифицирует — компилятор просто не знает про это, потому что все это не языковое средство.

Поэтому при **longjmp** в отличие от прочих регистровые переменные вернутся в исходное состояние (и избежать этого нельзя). Также компилятор обычно не знает, что вызов функции может привести к передаче управления в пределах данной функции. Поэтому в некоторых случаях он может не изменить значение переменной (например, полагая ее выходящей из области использования).

Модификатор **volatile** в данном случае поможет только тем переменным, к к которым он применён, поскольку он никак не влияет на оптимизацию работы с другими переменными...

Как получить адрес члена класса?

Поскольку указатель на член, в отличие от простого указателя, должен хранить также и контекстную информацию, поэтому его тип отличается от прочих указателей и не может быть приведён к **void***. Выглядит же он так:

```
int i; int f();
struct X int i; int f(); x,
*px = &x;
int *pi = &i; i = *pi;
int (*pf)() = &f; i = (*pf)();
int X::*pxi     = &X::i; i = x.*pxi;
int (X::*pxf)() = &X::f;
i = (px->*pxf)();
```

Зачем нужен for, если он практически идентичен while?

Уточним различие циклов **for** и **while**:

- **for** позволяет иметь локальные переменные с инициализацией;

- **continue** не «обходит стороной» выражение шага, поэтому

```
for(int i = 0; i < 10; i++) ... continue; ...
```

не идентично

```
int i = 0; while(i < 10) ... continue; ... i++
```

Зачем нужен NULL?

Формально стандарты утверждают, что **NULL** идентичен **0** и для обоих гарантируется корректное преобразование к типу указателя.

Но разница есть для случая функций с переменным числом аргументов (например, **printf**) — не зная типа параметров компилятор не может преобразовать **0** к типу указателя (а на писюках **NULL** может быть равным **0L**).

С другой стороны, в нынешней редакции стандарта **NULL** не спасёт в случае полиморфности: когда параметр в одной функции **int**, а в другой указатель, при вызове и с **0**, и с **NULL** будет вызвана первая.

Безопасно ли delete NULL? Можно ли применять delete[]var после new var[]? А что будет при delete data; delete data?

- **delete NULL** (как и **free(NULL)**) по стандарту безопасны;
- **delete[]** после **new**, как и **delete** после **new[]** по стандарту применять нельзя.

Если какие-то реализации допускают это — это их проблемы;

- повторное применение **delete** к освобождённому (или просто не выделенному участку) обладает «неопределённым поведением» и может вызвать всё, что угодно — **core dump**, сообщение об ошибке, форматирование диска и прочее;
- последняя проблема может проявиться следующим образом:

```
new data1; delete data1;
new data2;
delete data1; delete data2;
```

Что за чехарда с конструкторами? Деструкторы явно вызываются чаще…

На это существует неписанное «Правило Большой четвёрки»: если вы сами не озаботитесь о дефолтном конструкторе, конструкторе копирования, операторе

присваивания и виртуальном деструкторе, то либо Старший Брат озаботит вас этим по умолчанию (первые три), либо через указатель будет дестроиться некорректно (четвёртый).

Например:

```
struct String1 ... char *ptr; ... String1 &operator =
(String1&); ... ;
struct String2 ... char array[lala]; ... ;
```

В результате отсутствия оператора присваивания в **String2** происходило лишнее копирование **String2::array** в дефолтном операторе присваивания, поскольку **String1::operator =** и так уже дублировал строку **ptr**. Пришлось вставить.

Так же часто забывают про конструктор копирования, который вызывается для передачи по значению и для временных объектов. А ещё есть чехарда с тем, что считать конструктором копирования или операторами присваивания:

```
struct C0  C0 &operator = (C0 &src) puts("C0=");
return *this; ;
struct C1 :C0 C0 &
operator = (C0 &src) puts("C1="); return *this;
;
int main()
C1 c1, c2; c1 = c2;
```

Некоторые считают, что здесь должен быть вызван дефолтный оператор присваивания **`C1::operator=(C1&)'** (а не **`C1::operator=(C0&)'**), который, собственно, уже вызовет **C0::operator=(C0&)**.

Понадобилось написать метод объекта на ассемблере, а Ватком строит имена так, что это невозможно — стоит знак «:» в середине метки, типа xcvvxx:svvsvv. Какие ключи нужны чтобы он такого не делал?

```
class A;
extern "C" int ClassMetod_Call2Asm
(A*, ...);
class A
int Call2Asm(...) return ClassMetod_Call2Asm(this, ...);
;
```

Сожрет любой **Cpp** компилятор. Для методов, которые вы хотите вызывать из **asm** — аналогично...

Так можно произвольно менять генерацию имён в **Watcom:**

```
#pragma aux var "_*";
```

И **var** будет всегда генериться как **var**. А ещё лучше в данном случае использовать **extern «C»** и для переменных также.

После имени функции говорит о том, что Ватком использует передачу параметров в регистрах. От этого помогает **cdecl** перед именем функции.

Пример:

```
extern "C" int cdecl my_func();
```

Скажите почему возникает stack overflow и как с ним бороться?

Причины:

1. Велика вложенность функций

2. Слишком много локальных переменных (или большие локальные массивы);

3. Велика глубина рекурсии (например, по ошибке рекурсия бесконечна)

4. Используется **call-back** от какого-то драйвера (например, мыши);

В пунктах с 1 по 3 — проверить на наличие ошибок, по возможности сделать массивы статическими или динамическими вместо локальных, увеличить стек через **stklen (**для С++**)**, проверять оставшийся стек самостоятельно путем сравнения **stklen** с регистром **SP**.

В пункте 4 — в функции, использующей **call-back**, не проверять стек; в связи с тем, что он может быть очень мал — организовать свой.

Любители Каткома! А что, у него встроенного ассемблера нет что ли? Конструкции типа asm не проходят?

Встроенного **asm'a** у него на самом деле нет. Есть правда возможность писать **asm**-функции через **'#pragma aux ...'**.

Например:

```
#pragma aux DWordsMover = \
"mov esi, eax", \
"mov edi, ebx", \
"jcxz @@skipDwordsMover",      \
"rep movsd",     \
"@@skipDWordsMover:",   \
parm [ebx] [eax] [ecx] modify
[esi edi ecx]
```

```
void DWordsMover
(void* dst, void* src, size_t sz);
```

При создании 16-bit OS/2 executable Watcom требует либу DOSCALLS.LIB. Причем ее нет ни в поставке Ваткома ни в OS/2. Что это за либа и где ее можно достать?

Называют ее теперь по другому. В каталоге **LIB286** и **LIB386** есть такая **OS2286.LIB**. Это то, что вам нужно. Назовите ее **DOSCALLS.LIB** и все.

BC не хочет понимать метки в ассемблерной вставке — компилятор сказал, что не определена эта самая метка. Пришлось определить метку за пределами ASM-блока. Может быть есть более корректное решение?

Загляните в исходники **RTL** от С++ 3.1 и увидите там нечто красивое.

Например:

```
#define I asm
//........
I or si,si
I jz m1
I mov dx,1 m1:
I int 21h
```

и т.д.

Есть — компилировать с ключом **'-B' (via Tasm) aka '#pragma inline'**. Правда, при этом могут возникнуть другие проблемы: если присутствуют имена **read** и **_read** (например), то компилятор в них запутается.

Было замечено, что Борланд (3.1, например) иногда генерит разный код в зависимости от ключа **-B**.

Как правило, при его наличии он становится «осторожнее» — начинает понимать, что не он один использует регистры.

Почему при выходе из программы под BC++ 3.1 выскакивает «Null pointer assignment»?

Это вы попытались что-то записать по нулевому адресу памяти, чего делать нельзя.

Типичные причины:

- используете указатель, не инициализировав его.

Например:

```
char *string; gets(string);
```

- запрашиваете указатель у функции, она вам возвращает **NULL** в качестве ошибки, а вы этого не проверяете.

Например:

```
FILE *f = fopen("gluck", "w"); putc('X', f);
```

Это сообщение выдаётся только в моделях памяти **Tiny**, **Small**, **Medium**.

Механизм его возникновения такой: в сегменте данных по нулевому адресу записан борландовский копирайт и его контрольная сумма. После выхода из **main** контрольная сумма проверяется и если не совпала — значит напорчено по нулевому адресу (или рядом) и выдаётся сообщение.

Как отловить смотрите в **HELPME!.DOC** — при отладке в **Watch** поставить выражения:

```
*(char*)0,4m
(char*)4
```

потом трассировать программу и ловить момент, когда значения изменятся.

Первое выражение — контрольная сумма, второе — проверяемая строка.

При запуске программы из BC (Ctrl-F9) все работает нормально, а если закрыть BC, то программа не запускается. Что делать?

Если вы используете **BWCC**, то эту либу надо грузить самому — просто среда загружает **BWCC** сама и делаёт её доступной для программы. Советуем вам пользоваться таким макросом:

```
#define _BEST EnableBWCC(TRUE); \
EnableCtl3d(TRUE); \
EnableCtl3dAutosubclass(TRUE)
```

Потом в **InitMainWindow** пишете:

```
_BEST;
```

и все будет хорошо.

Вообще-то правильнее **OWL**'евые экзепшены ловить и выдавать сообщение самостоятельно. Заодно и понятнее будет отчего оно произошло:

```
int OwlMain(int /*argc*/, char* /*argv*/[])
int res;
TRY res = App().Run();
CATCH((xmsg &s)
//Какие хочешь ексепшены
MessageBox(NULL, "Message", s.c_str());
return res;
```

Почему иногда пытаешься проинспектировать переменную в BC++ во время отладки, а он ругается на inactive scope?

Вот пример отлаживаемой программы. Компилим так:

```
bcc -v is.cpp
=== Cut ===
#include <iostream.h>
void a()
int b = 7;
cout << b << endl;
void main()
a();
=== Cut ===
```

Входим в **TD**. Нажимаем **F8**, оказываемся на строке с вызовом **a()**. Пытаемся **inspect b**. Естественно, не находим ничего. А теперь перемещаем курсор в окне исходника на строку с **cout**, но трассировкой в **a()** не входим и пробуем посмотреть **b**. И вот тут-то и получаем **inactive scope**.

Были у меня две структуры подобные, но вторая длиннее. Сначала в функции одна была, я на ней отлаживался, а потом поменял на вторую, да только в malloc'е, где sizeof(struct ...) старое оставил, и налезали у меня данные на следующий кусок хипа

Для избегания подобной баги можно в **Си** сымитировать Сиплюсный **new**:

```
#define tmalloc(type)  ((type*)malloc(sizeof(type)))
#define amalloc(type, size) ((type*)malloc(sizeof(type) *
(size)))
```

Более того, в последнем **define** можно поставить **(size) + 1**, чтобы гарантированно избежать проблем с завершающим нулём в строках.

Можно сделать иначе. Поскольку присвоение от **malloc()** как правило делают на стилизованную переменную, то нужно прямо так и писать:

```
body = malloc(sizeof(*body));
```

Теперь вы спокойно можете менять типы не заботясь о **malloc()**. Но это верно для Си, который не ругается на присвоение **void*** к **type*** (иначе пришлось бы кастить поинтер, и компилятор изменения типа просто не пережил бы).

Вообще в Си нет смысла ставить преобразования от **void*** к указательному типу явно. Более того, этот код не переносим на C++ — в проекте стандарта C++ нет **malloc()** и **free()**, а в некоторых компиляторах их нет даже в **hosted c++** заголовках.

Проще будет:

```
#ifdef __cplusplus
# define tmalloc(type) (new type)
# define amalloc(type, size)
(new type[size])
#else
# define tmalloc(type) malloc(sizeof(type))
# define amalloc(type, size) malloc(sizeof(type) * (size))
#endif
```

Суммируя вышеперечисленное, можно отметить следующее. Необходимо скомбинировать все варианты:

```
#ifdef __cplusplus
# define tmalloc(type) (new type)
# define amalloc(type, size)
(new type[size])
# define del(var)      delete(var)
#else
# define tmalloc(type) ((type*)malloc(sizeof(type)))
# define amalloc(type, size) ((type*)malloc(sizeof(type) *
(size)))
# define del(var)      free(var)
# define vmalloc(var)
((var) = malloc(sizeof(*(var))))
#endif
```

Я не понимаю, почему выдаются все файлы, вроде указал, что мне нужны только с атрибутом директория? Можно, конечно, проверять ff_attrib, что нашли findfirst и findnext, но это мне кажется не выход. Может я что не дочитал или не понял?

```
done = findfirst("*.*", &onlydir, FA_DIREC);
while(!done)
   cout << onlydir.ff_name << endl;
```

```
done = findnext(&onlydir);
```

Это не баг, это фича MS DOS. Если атрибут установлен, то находятся как файлы с установленным атрибутом, так и без него. Если не установлен, то находятся только файлы без него. И проверять **ff_attrib** вполне выход. Вы не дочитал хелп про **findfirst/findnext**.

Создается файл: fopen(FPtr, "w"). Как может случиться, что структура пишется на диск некорректно?

```
fopen (FPtr,"wb");
```

Режим не тот...

При печати функцией cprintf в позицию экрана x = 80, y = 25 происходит автоматический перевод строки (сдвиг всего экрана на строку вверх и очистка нижней строки) и это знакоместо так и остается пустым. Может кто знает, как вывести символ в это знакоместо?

Нажмите **Ctrl+F1** на слове **_wscroll** в Борландовском **IDE**. Правда, **printf** это не вылечит, так как его вывод идёт не через борландовскую библиотеку.

Как очистить текстовый экран в стандарте ANSI C?

Никак, в ANSI C нет понятия экрана и текстового режима. В Turbo Си так:

```
#include <conio.h>
void main(void) clrscr();
```

Можно также попробовать выдавать ANSI **ESC**-коды или сделать следующее:

```
#include <stdio.h>
#define NROWS 2*25 /*
```

Чтобы обработать случай курсора в первой строке:

```
void main(void)
short i;
for(i = 0; i < NROWS; i++) puts("");
```

Но это совершенно негарантированные способы.

Используя прерывания VESA, пытаюсь подключить мышь и вот тут начинается сумасшедший дом... Что делать?

Мышиный драйвер не знает какой у вас на данный момент видео-режим и использует параметры предыдущего режима (у вас он наверное текстовый — там мышь скачет дискретно по 8).

Поэтому, рисовать мышь вы должны сами. А чтобы координаты мыши отслеживать, у **33h** прерывания есть функция, которая возвращает смещение мыши от последней ее позиции.

Можно обойтись без рисования своего курсора мыши если найти драйвер, понимающий **VESA**-режимы.

Например, в **Logitech MouseWare 6.3** входит некий оверлейчик для генерации курсора для режимов Везы, который соответствует какой-то там совместной спецификации Везы и Логитеча.

Как установить патчи на версию «Try & Bye»?

Для Win32 в реестре меняете ключ:

```
HKEY_LOCAL_MACHINE\SOFTWARE\IBM\IBM VisualAge for C++ for
Windows Demo\demo
```

на

```
HKEY_LOCAL_MACHINE\SOFTWARE\IBM\IBM VisualAge for C++ for
Windows\3.5
```

Для OS/2 редактируете файл **\os2\system\epfis.ini** при помощи любого редактора **INI** файлов и заменяете в нем:

- имя апликации

```
EPFINST_IBM VisualAge C++ for OS/2_TRIAL_COPY_0001
```

  или что-то подобное на

```
EPFINST_IBM VisualAge C++ for OS/2_5622-679_0001
```

- содержимое ключа **ApplicationName** для данной апликации изменяете с

```
IBM VisualAge C++ for OS/2 TRIAL COPY
```

  или опять что-то подобное на

```
IBM VisualAge C++ for OS/2
```

- файл **cppexit.dll** копируете в **exit.dll**.

После таких манипуляций можно спокойно ставить патчи.

Как сортировать записи в IVBContainerControl?

IVBContainerControl отвечает только за отображение. Капать надо в области **IVSequence**, на который есть ссылка в объекте **IVBContainerControl**.

Он ведь только то отображает, что в **IVSequence * IVBContainerControl::items** содержится. Так что берете этот **items** и сортируете.

Для создания невизуальных part лучше использовать VB или .VBE?

Настоятельно рекомендуется **.VBE**

Где находятся описания типов (не классов) для VB?

.VBE, использовать редактор **Part** для описания типов нельзя. Правильнее всего посмотреть **.\Samples\VisBuild\vbSample*.VBE** Там хорошо показано, как делать описание блоков функций, типов и перечислений.

Что можно использовать для выбора цвета?

Для выбора цвета лучше всего использовать **..\Sample\VisBuild\Doodle\ ClrDlg.VBB.**

Можно ли использовать VAC++ без WPS и WF?

Можно. Надо инсталлировать его из под **WPS**, а потом заменить его на что-нибудь типа **FileBar**.

Будет работать все, кроме редактора. Это позволяет использовать **VB** на **16 MB**.

Есть некое окошко, которое должно делать нечто через каждые N секунд. Как это правильно изобразить в VisualBilder/PartEditor?

На **Ibm**'ком сервере в примерах по **VAC++** лежит как раз подобный пример. Файл **vbtimer.zip** размером ~30 K.

Я уже замучился загружать все .vbb модули в Visual Builder. Что делать?

Создайте файлик **VbLoad.Dat** со списком этих файлов с указанием пути и положите его либо в каталог, где живут файлы приложения, в случае если **Visual Builder** запускается оттуда, либо (что подходит только для одного проекта) в каталог в **VbBase.Vbb**, **VbDax.Vbb e.t.c** (он называется **IVB** для **Win** и **DDe4Vb** для **Os/2**).

Пути указывать не обязательно, если каталог, где они лежат «входит» в переменную окружения **VBPATH**.

Где взять документацию на Ватком?

В поставке. Все что есть в виде книжек включено в дистрибутив, кроме книги Страуструпа.

Как поставить Ватком версии 10 под пополамом, при установке в самом конце происходят странные вещи?

Лучше всего провести установку (копирование файлов и создание каталогов) в досовской сессии, а потом пополамным инсталлером просто откорректировать конфиги и создать все необходимые установки.

В русифицированной WIN95 криво, устанавливается WATCOM. Не создает папки со своими иконками. Что делать?

Нужно сделать каталоги **\Windows\Start Menu\Programs** и переустановить Ватком. Потом перекинуть **.lnk** куда вам нужно.

При отсутствии нужных англоязычных папок ссылки улетают в никуда.

Он занял очень много места на диске, от чего можно избавиться?

Если вы не предполагаете писать программы под какие-либо платформы, то не стоит устанавливать и библиотеки для них, если вы собираетесь работать под пополамом, можно смело прибить досовские и виндовозные хелпы, и программку для их просмотра.

Кроме того, надо решить какой средой вы будете пользоваться, компилировать в дос-боксе или нет.

Пополамный компилятор ресурсов под досом очень слаб и сваливается по нехватке памяти даже на простых файлах. Более того есть мнение, что при компиляции в осевой сессии, по крайней мере линкер работает примерно в 3 раза быстрее.

Вот я его поставил, ничего не понятно, с чего начать?

Прежде всего — почитать документацию, версия 10 поставляется с огромными файлами хелпа, если вы работаете под пополамом — используйте **VIEW** или иконки помощи в фолдере, если под **Windows** — соответственно программку **WHELP** для просмотра ***.HLP**, ну и под досом — аналогично, правда там вы не получите красивых окошек и приятной гипертекстовой среды.

Где у него IDE, я привык, чтобы нажал кнопку, а оно откомпилировалось?

IDE существует, но работает только под **Windows** или **OS/2**. Для работы в Досе используйте командную строку.

Если вы так привыкли к **IDE** — поддержка Ваткома есть в **MultiEdit**, и комплект удобных макросов тоже.

С чего начать, чтобы сразу заработало?

Начните с простейшего:

```
#include <stdio.h>
main()
puts("Hello, world");
```

Для компиляции нужно использовать:

```
wcl hello.c – для DOS/16
```

```
wcl386 /l=dos4gw hello.c – для DOS4GW
```

Я написал простейшую программку, а она внезапно повисает, или генерирует сообщение о переполнении стека, что делать? В то же время когда я компилирую эту программку другим компилятором — все работает нормально?

Желательно сразу после установки поправить файлики **WLSYSTEM.LNK**, поставив требуемый размер стека, по умолчанию там стоит 1 или 2 Кб, чего явно недостаточно для программ, создающих пусть даже небольшие объекты на стеке.

Для большинства применений достаточно размера стека в 16 или 32 килобайта. Если вы работаете под экстендером, можно поставить туда хоть мегабайт.

Я столкнулся с тем, что Ватком ставит знак подчеркивания не в начало имени, а в конец, к чему бы это?

Положение знака подчеркивания говорит о способе передачи параметров в данную функцию, если его нет совсем, параметры передаются через регистры, если сзади — через стек.

Я написал подпрограмму на ассемблере, со знаком подчеркивания спереди, а Ватком ищет то же имя, но со знаком «_» сзади, как это поправить?

Можно написать:

```
#pragma aux ASMFUNC "_*";
```

и описывать все свои функции как:

```
#pragma aux (ASMFUNC) foo;
#pragma aux (ASMFUNC) bar;
```

Причем, есть специальное значение — символ «^», который сигнализирует, что имя надо преобразовать в верхний регистр, например: **#pragma aux myfunc «^»;** приведет к появлению в объектном файле ссылки на «**MYFUNC**».

Есть библиотека, исходники которой отсутствуют, как заставить Ватком правильно понимать имена функций и ставить знак «_» спереди а не сзади?

Нужно в файле заголовка описать данные функции как **cdecl**, при этом параметры будут передаваться через стек и имя функции будет сформировано правильно.

Как сделать так, чтобы в некоторых случаях Watcom передавал параметры не через регистры, а через стек?

Использовать **cdecl**.

Например:

```
extern void cdecl dummy( int );
```

Как делать ассемблерные вставки в программу?

Примерно так:

```
unsigned short swap_bytes ( unsigned short word );
#pragma aux swap_bytes = "xchg ah,
al" \
parm [ ax ] \
value [ ax ];
```

Слово **parm** определяет в каком регистре вы передаете значение, слово **value** — в каком возвращаете.

Можно использовать метки. Есть слово **modify** — можно указать что ваша вставка (или функция) не использует память, а трогает только те или иные регистры.

От этого оптимизатору лучше жить. Прототип не обязателен, но если есть, то компилер проверяет типы.

Надо слепить задачу под графику, но нужны окошки и мышь. Тащить ли ZINC 3.5, или в графике описать что-нибудь свое. Может, под Ватком что-то есть более мощное и готовое?

Ничего лучше **Зинки** пока нет. Тащите лучше **Зинку 4.0**, она вроде под Ватком лучше заточена.

При написании некоторых функций по видео-режимах вдруг захотелось мне сотворить динамические библиотеки. Есть мысля генерить exe-файл а затем грузить его. Что делать?

Использовать **DOS4GW/PRO**. Он вроде поддерживает **DLL**. Или пользоваться **PharLap TNT**, он тоже поддерживает.

Грузить экзешник тоже можно, но муторно. Через **DPMI** аллоцируете сегмент (сегменты) делаете из них код и данные, читаете экзешник и засовываете код и данные из него в эти сегменты. Лучше использовать **TNT**.

Графическая библиотека Ваткома отказывается переключать режимы/банки или делает это криво. Что делать?

В результате ковыряния в библиотеке выяснилось, что криворукие ваткомовцы совершенно не задумываются ни о какой переносимости и универсальности их библиотек.

В результате, если видео-карта имеет в биосе прошитое имя производителя или другую информацию о нем, то для нее будет вызываться вместо функции переключения банков через **VESA**,

другая функция, работающая с картой напрямую (иногда даже через порты).

Единственная проблема, что у каждого производителя рано или поздно выходят новые и продвинутые карты, раскладка портов в которых может отличаться от той, которая использовалась в старых моделях.

В результате, все это свинство начинает глючить и иногда даже виснуть.

После того, как вы руками заткнете ему возможность использовать «родные» фишки для конкретной карты и пропишите пользоваться только **VESA** — все будет работать как из пушки.

Как затыкать — а просто, есть переменная: **_SVGAType**, которая описывается следующим образом:

```
"extern "C" int _SVGAType;",
```

и потом перед (важно!) вызовом **_setvideomode** нужно сказать:

```
"_SVGAType = 1;"
```

Как руками слинковать exe-файл?

Командой **WLINK**, указав параметры.

```
name ...
system ...
debug all
option ...
option ...
option ...
...
file ...
file ...
...
libpath ...
library ...
```

Например:

```
wlink name myprog system dos4gw debug all file myprog
```

Что такое ms2vlink и зачем она нужна?

Это для тех кто переходит с мелкософтовского **Си.** Преобразователь команд **LINK** в **WLINK**.

Что такое _wd_?

Это отладчик, бывший **WVIDEO**, но с более удобным интерфейсом.

Поставляется начиная с версии 10.

Нужно состряпать маленький NLM'чик. Что делать?

Вам нужен **WATCOM 10.0**. В него входит **NLM SDK** и вроде хелп к нему. Если **WC <= 9.5,** то нужен сам **NLM SDK** и документация.

```
// Линковать :
// (файл wclink.lnk например)
// system netware
// Debug all
// opt scr 'Hello, world'
// OPT VERSION=1.0
// OPT COPYR 'Copyright (C) by me, 1994'
#include <conio.h>
void main( void )
cprintf( "Hello, world!\n\r" );
ConsolePrintf( "Hello, World - just started!\n\r" );
RingTheBell();
```

Собираю программу под OS/2 16-бит, линкер не находит библиотеку DOSCALLS.LIB. Кто виноват и что делать?

Никто не виноват. В поставке Ваткома есть библиотека **os2286.lib**.

Это она и есть. Ее надо либо переименовать в **doscalls.lib**, либо явно прилинковывать.

Что такое удаленная отладка через pipe? Как ею пользоваться под OS/2?

В одной сессии запускается **vdmserv.exe**, потом запускается отладчик **wd /tr=vdm** и соединяется с **vdmserv** по пайпу, ну и рулит им. Как удаленная отладка через компорт работает знаете? Вот тут так же, только через пайп.

Собираю 32-битный экзешник под PM с отладочной информацией (/d2), но после того как осевым rc пришпиливаю к нему ресурсы, отладочной информации — как не бывало. Это лечится как-нибудь?

Откусываете дебугинфу **wstrip**'ом в **.sym** файл и потом присобачиваете ресурсы. Если имя экзешника и имя **.sym** совпадают, дебаггер сам его подхватит.

Отладочную информацию надо сбрасывать в **SYM**-файл:

```
wcl386 /d2 /"op symf" /l=os2v2_pm
```

WATCOM на 4 мб компилирует быстрей чем на 8 мб, а на 8 мб быстрее чем на 16 мб, почему?

Чем больше памяти, тем лучше работает оптимизатор. Можно дать ему фиксированный размер памяти — **SET WCGMEMORY=4096**, и тогда он не будет пользоваться лишней памятью.

Учтите, что для компиляции программ для Windows на C++ данного значения может не хватить.

Есть такая штука — pipe в gcc и bcc. А вот в Watcom'e как перехватить выхлоп программы?

В смысле забрать себе **stdout** и **stderr**? Да как обычно — сдупить их куда-нибудь. Функцию **dup()** еще никто не отменял.

А есть ли способ перехватить ошибку по нехватке памяти? То есть какой-нибудь callback, вызываемый диспетчером памяти при невозможности удовлетворить запрос?

В C++ есть стандартный: **set_new_handler()**.

Чем отличаются статические DLL от динамических?

Разница в том, что вы можете функи из **DLL** на этапе линковки в **EXE**'шник собрать (**static**). А можете по ходу работы проги **DLL** грузить и функи выполнять (**dynamic**).

Решил тут DLL под OS/2 создать — ничего не вышло. Что делать?

Вы динамически собираетесь линковать или статически? Если статически, тогда вам просто **declare func** сделать и включить **dll** в **test.lnk**.

Если динамически, то вы должны прогрузить **dll**, получить адрес функи и только после этого юзать. Можно делать это через **API OS/2: DosLoadModule DosQueryProcAddr DosFreeModule.**

Например:

```
exe.c:
#define INCL_DOSMODULEMGR
#include <os2.h>
#define DLLNAME "DLL"
PFN Dllfunc;
char FuncName[]="RegardFromDll_";
char LoadError[100];
void main()
```

```
HMODULE MHandle;
DosLoadModule( LoadError, sizeof( LoadError ), DLLNAME,
&MHandle ); DosQueryProcAddr( MHandle, 0, FuncName, &Dllfunc
); (*Dllfunc)();
DosFreeModule( MHandle );
dll.c
#include <stdio.h>
void RegardFromDll( void )
printf( "This printed by function, loaded vs DLL\n" );
```

Когда компиляете свою **DLL**, то добавьте свич **-bd**, который создаст в **.obj** такое дело, как **DLLstart**. После этого все заработает:

```
wpp386 -bd -4s -ox dll.cpp
```

Как подавить варнинги о неиспользованных аргументах?

Если используете плюсовый компилер — просто опускайте имена параметров, например:

```
void foo( int bar, char* )
```

или

```
#pragma off(unreferenced)
```

Можно использовать макрос:

```
#ifndef NU
#ifdef __cplusplus
#ifdef __BORLANDC__
#define NU( ARG ) ARG
#else
#define NU( ARG ) (void)ARG
#endif
#else
#define NU( ARG ) ARG=ARG
#endif
#endif
```

Для многих, особенно юниксового происхождения, компайлеров работает **/*ARGSUSED*/** перед определением функции.

Подскажите, как в watcom'е увеличить число открытых файлов?

Смотрите **TFM**. **_grow_handles(int newcount)**.

Как заставить 16-ти битные OS/2 задачи видеть длинные имена файлов?

Опция **newfiles** для линкера.

Что-то у меня Dev.Toolkit for OS/2 Warp кВаткому WC10.0 прикрутить не получается. Говорит definition of macro '_Far16' not identical previos definition. Что делать?

Воткните где-нибудь определение **IBMCPP** или **-D_IBMCPP** в командной строке или **#define** перед **#include <os2.h>**.

Для Ваткома 10.5a надо не просто **d_IBMCPP**, а **-d_IBMCPP_=1.**

Пишу: printf («*»);, а он сразу ничего не печатает. Что делать?

В стандарте **ansi**, чёткого определения как должны буферизовываться потоки **stdin/stdout/stderr** нет, нормальным является поведение со строчной буферизацией **stdout/stdin**.

Всякие другие дос-компилеры обычно не буферизуют **stdout** совсем, что тоже нормально. Признаком конца строки в потоке является **'\n'**, именно при получении этого символа происходит **flush** для **line buffered** потока.

Выходов два: отменить буферизацию или писать **'\n'** в нужных местах. Можно **fflush(stdout)** звать, тоже вариант.

Буферизация отменяется **setbuf(stdout,NULL)** или **setvbuf(stdout,NULL,_IONBF,0)**.

Можно ли сделать встроенный в exe-шник DOS4GW, как в DOOM?

Легально — нет. Предыдущие версии позволяли просто скопировать:

```
copy /b dos4gw.exe + a.exe bound.exe
```

Но сейчас (начиная с версии 10.0a) это не работает и для этой цели нужно приобрести **dos4gw/pro** у фирмы **Tenberry Software**.

Нелегально — да. Существует утилита **dos4g/link** для автоматизированного выдирания и вклеивания экстендеров из/в **EXE-**файлов. Помещалась в **WATCOM.C** в **uuencode** и доступна от автора.

Нужно взять не тот **DOS4GW**, что в комплекте (**DOS4GW 1.97**), а **Pro**-версию (**DOS4GW Professional**). Выдрать можно из **DOOM**, **HERETIC**, **HEXEN**, **WARCRAFT2** и т.д., где он прибинден. Причем можно найти 2 разновидности **Pro 1.97** — одна поддерживает виртуальную память, другая нет и еще что-то по мелочи.

Различаются размерами (который с виртуалкой — толще). Прибиндить можно разными тулзами, например **PMWBIND** из комплекта **PMODE/W**.

Также можно отрезать у **dos4gw.exe** последние несколько байт с хвоста, содержащие строку **WATCOM patch level [...]**. Далее обычным бинарным копированием:

```
copy /b dos4gw.exe myexe.exe mynewexe.exe
```

Работоспособно вплоть до версии **DOS/4GW 1.95**. В версии **1.97** введена проверка на внедренность **linexe** в хвост экстендера.

Еще существует родной биндер для **DOS/4GW**. Он в какой-то мере может помочь **pmwbind.exe** от **PMODE/W** (однако версия **1.16** не понимала каскадный формат **DOS/4GW**, работоспособна для одномодульного **4GW/PRO**); решает проблему тулза **dos4g/link**, которая доступна у автора или у модератора.

Рекомендуется попробовать **4GWPRO** (выдрать из игрушек с помощью **pmwbind.exe** или **dos4g/link**), усеченный вариант **DOS/4GW** (в модулях **4grun.exe**, **wd.exe —** для ДОС), а также **PMODE/W**.

В поставке **DOS4G** есть **4GBIND.EXE** (но для этого надо купить или украсть **DOS4G**).

Как определить количество свободной памяти под dos4gw? Попытки использовать _memavl и _memmax не дают полную картину. Что делать?

Количество свободной памяти под экстендером — не имеет смысла, особенно если используется своппинг. Для определения наличия свободного **RAM** нужно использовать функции **DMPI**, пример использования есть в хелпе.

Как добраться до конкретного физического адреса под экстендером?

Вспомните про линейную адресацию в **dos4gw**. Он в этом плане очень правильно устроен — например, начало сегмента **0xC000** находится по линейному адресу **0x000C0000**.

Вот примерчик, который печатает сигнатуру **VGA** биоса.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
void main( void )
```

```
unsigned i;
for( i = 0; i < 256; i++)

char c;
c = *(char*)( 0x000C0000 + i );
putchar( isprint( c ) ? c: '.' );
```

Как отлаживать программы, работающие в Pharlap режиме?

```
wd /tr=pls file.exe,
```

для фарлапа или

```
wd /tr=rsi file.exe
```

для **dos4gw.**

Какая разница между dos4gw и Pharlap? Или это одно и то же?

Это разные экстендеры. Самая существенная для вас разница — **dos4gw** входит в поставку Ваткома, а фарлап — нет.

Что такое RUN386.EXE? Вроде с его помощью можно пускать фарлаповые exe-шники?

Это рантайм фарлапа. Но он денег стоит.

DOS4GW такой огромный (больше 200 к), что можно использовать вместо него, чтобы поменьше диска занимало?

Существует шароварный экстендер **PMODE**, у которого есть версия, рассчитанная на **Watcom** — **PMODE/W**, ее можно использовать вместо **DOS4GW**, она занимает всего 9 к, встраивается внутрь **exe**-файла.

Он не обрабатывает некоторые исключительные ситуации, поэтому отлаживать программу все-таки лучше с **dos4gw**, а встраивать **pmode/w** лишь в окончательный вариант.

100% совместимости с **dos4gw** никто, конечно, гарантировать не может, но говорят, что под ним удалось запустить даже **D00M.**

Более того, 100% совместимости просто нет — например, графические программы под **DOS/4G** в Цинке, которые определяют наличие **DOS/4GW** путем вызова **int 21h, ah = 0xFF**.

При этом **DOS/4GW** и **PMODE/W** возвращают различный (хотя и похожий) результат.

А также **DOS/4G** «подобные»:

- **WDOSX** (последняя версия 0.94, size ~12 Kb)
- **DOS32A** (последняя версия 4.30, size ~20 Kb)

В чем отличия между DOS4GW и DOS4GW PRO?

DOS4GW:

- используется в виде отдельного **.EXE** модуля, имеет ограничения по размеру виртуальной памяти (16 Мб), ограничение по общей используемой памяти (32 Мб)
- отсутствует поддержка некоторых **DPMI** вызовов (например **303h — allocate callback**)
- отсутствует возможность писать **TSR'**ы
- отсутствует поддержка **DLL**, freeware
- 4GWPRO — встраивается в исполняемую программу
- ограничений в размере виртуальной памяти нет
- полная поддержка **DPMI 1.0**
- поддержка **DLL**
- поддержка **TSR**
- стоит денег.

DOS4G:

- не привязан к конкретному компилятору
- возможен запуск нескольких **.EXE'**шников под одним экстендером
- поддержка **DLL** документирована
- обильная документация
- стоит больших денег.

В процессе экспериментов выяснилось, что поддержка виртуальной памяти (**VMM — virtual memory manager**) и поддержка полного набора **DPMI** вызовов присутствуют не во всех вариантах **4GWPRO**.

Можно ли поиметь 4GWPRO даром?

Да, можно. Для этого его надо «вырезать» из головы программы собранной с **4GWPRO**. Обычно такая программа при запуске сама об этом сообщает.

Однако не из любой программы можно получить полноценный экстендер.

Ниже приведен список программ подвергшихся обрезанию и результаты.

- **ACMAIN.EXE**,
- **DESCENT.EXE**,
- **HB5.EXE**,
- **HEROES.EXE**

дают версию **1.97** с полным набором прелестей. Размер: 217764 байта.

- **ABUSE.EXE**,
- **BK.EXE**,
- **HEXEN.EXE**,
- **ROTT.EXE**,
- **TV.EXE (Terminal Velocity)**

дают версию **1.97** без **VMM** и поддержки расширенного набора **DPMI**.

Размер: 157268 байт.

- **ACRODOS.EXE (Acrobat reader for DOS)**

дает версию **1.97** с **VMM,** но без расширенного набора **DPMI**.

Размер: 203700 байт.

- **D4GRUN.EXE** (из **Watcom 10.0a)**

дает версию **1.96** без **VMM,** но с расширенной поддержкой **DPMI** (но судя по надписям внутри — это **DOS4G,** а не **4GWPRO**). Размер: 154996 байт.

- **DOOM2.EXE**

дает версию **1.95** без поддержек **VMM** и расширенного набора **DPMI**. Размер: 152084 байт.

Как переделать программу, скомпилированную под DOS4GW для использования с полученным 4GWPRO?

```
COPY /B 4GWPRO + OLD.EXE NEW.EXE
```

Почему полученный 4GWPRO не дает использовать VMM, или не дает больше 16 Мб?

Простое шаманство поможет:

```
00000247  бит   0-3   ???
00000247: бит   4     1-VMM по умолчанию вкл., 0-выкл.
00000247: бит   5     ???
00000247: бит   6     1-подавлять заставку при старте
00000247: бит   7     ???
```

Для **1.97** размером 217764 байта.

0001BFF8: (4 байта) размер виртуальной памяти по умолчанию.

Можно ли использовать DLL с DOS4GW?

Можно, это обеспечивает утилита **DLLPOWER**.

Ищите в **SimTel**'овских архивах файлы **dllpr251.zip**, **dllpr254.zip** и может быть уже есть более поздние.

Я всю жизнь писал на Борланд-С, теперь решил перебраться на Ватком, как мне проще всего это сделать?

Перенос ваших программ возможен, но скорее всего вам придется править ваш код. Начать можно с изменения **int -> short.**

Ватком ругается на стандартные библиотечные функции, в то время как ВС жует их нормально, что делать?

Большинство программ, которые нормально работали под **ВС**, будут нормально компилироваться и Ваткомом, нужно лишь немного изменить код, использующий специфичные функции, реализованные в **ВС**, как расширение стандартной библиотеки.

Для большинства таких функций есть аналогичная или подобная функция.

Например, вместо **gettime()** следует использовать **dos_gettime()**, вместо **findfirst — dos_find_first**, и так далее.

Обратитесь к хелпу по **ВС**, вы наверняка найдете имена аналогичных функций, запросив помощь по тем именам, которые не устроили компилятор Ваткома.

Кроме того, следует помнить, что например **random(num)** не является стандартной функцией, а это просто макрос из **stdlib.h**, и вместо него можно использовать конструкцию типа (**rand() % num**).

Можно ли перенести под Ватком программы, написанные с применением OWL или TVision?

OWL — скорее всего нет, поскольку он построен на расширениях синтаксиса, которые не обрабатывается компилятором Ватком. Для переноса **TVision** есть несколько вариантов.

Существуют **diff**ы для преобразования **TV** под **GNU C++** (продукт называется **GVISION**. Это не решает разом все проблемы, естественно, но по крайней мере этот вариант **TV** заточен под 32 бита и флат модель.

Совсем недавно стали доступны два порта **TV** под **Watcom** C++.

Первый — это перенос под полуось и **dos4gw TV 1.0**.

Второй имеет внутри маленькую доку и собственно сырец+**makefile**. Доку рекомендуем внимательно прочесть — может пригодится при перекомпиляции ваших программ.

В моей программе используются inline ассемблер и псевдорегистры, смогу ли я заставить их работать под Ваткомом?

Нет. Придется переписать на встроенном ассемблере, либо вынести в отдельный **.asm** модуль.

С 11 версии поддерживает стиль a la Borland:

```
_asm ...
```

А нельзя ли как-нибудь на ваткоме реализовать _AX из Borland? А то переношу под него библиотеку, а там они активно юзаются

Если вам **_AX** нужен на **read-only**, то можно сделать прозрачно:

```
=== Cut ===
short reg_ax ( void ) ;
#define _AX     reg_ax()
#pragma aux     reg_ax = \
value [ax]
#if defined( __386__ ) || defined( __FLAT__ )
int      reg_eax ( void ) ;
#define _EAX reg_eax()
#pragma aux     reg_eax = \
value [eax]
#define  rAX reg_eax() // чтобы не задумываться о контексте
#else
```

```
#define   rAX  reg_ax()
#endif
=== Cut ===
```

А если для модификации, то лучше не полениться и сделать как просит Ватком (то бишь оформите этот фрагмент как **inline-asm**).

Встречается проблема, когда надо собирать смешанный проект — часть модулей компилится Ваткомом, а часть Борландом (например ассемблером). Линкер падает по трапу, вываливается с бредовыми ошибками и вообще ведет себя плохо. Что делать?

На худой конец есть способ:

- борландовый

```
obj->wdisasm->.asm->wasm->
```

- ваткомовский

```
obj
```

А дальше это отдавать как обычно **wlink'у**.

Есть еще народное средство — нужно взять **tasm 3.2**, а еще лучше **tasm 4.0**, последний хорош тем, что имеет режимы совместимости по синтаксису со всеми своими предками...

Или **TASM32 5.0** с патчем (обязательно 32 bits)

При задании надписей заголовков окон, меню, кнопок и т.п. на русском языке все прекрасно видно пока я нахожусь в режиме дизайнера. Стоит только запустить созданную аппликуху — кириллица исчезает напрочь.

Замените **WRC.DLL** из поставки **Optima 1.0** на **WRC.EXE** из поставки **Watcom 11.0** и все придет в норму.

Какой компилятор C (C++) лучше всех? Что лучше: Watcom C++ или Borland C++? Посоветуйте самый крутой компилятор?!

Смотря для чего. Если нужна многоплатформенность и хорошая кодогенерация, то лучше — Ватком. Но следует учитывать, что производство Ваткома прекращено и компилятор уже не соответствует в полной мере стандарту C++, и уже никогда не будет соответствовать. А вот для разработки приложений под Win32 лучше будет Borland C++ Builder, хотя качество кодогенерации у него ниже и ни о какой многоплатформенности говорить не приходится. Что же касается BC++ 3.1 — это не более, чем учебный компилятор, и он уже давно забыт.

Не следует забывать и о **gcc**, который есть подо все мыслимые платформы и почти полностью поддерживает стандарт, однако он (точнее, его Win32 версия — cygwin) не слишком удобна для создания оконных Win32 приложений, с консольными — все в порядке, причем консольные приложения можно создавать и досовой версией — **djgpp**, дополненной пакетом **rsxnt**.

А вообще — рассуждать, какой компилятор лучше, достаточно бесполезное занятие. Идеального компилятора не существует в природе. Совершенно негодные компиляторы не имеют широкого распространения, а тот десяток компиляторов (не считая специфических кросс-компиляторов под разные встроенные системы), которые имеют широкое распространение, имеют свои достоинства и недостатки, так что приходится подбирать компилятор, наиболее подходящий для решения конкретной задачи.

Есть ли в Watcom встроенный ассемблер?

Встроенного asm'a у него на самом деле нет. Есть правда возможность писать asm-функции через **#pragma aux ...**. Например:

```
#pragma aux DWordsMover =        \
   "mov esi, eax",        \
   "mov edi, ebx",        \
   "jcxz @@skipDwordsMover", \
   "rep movsd",           \
   "@@skipDWordsMover:",         \
   parm [ebx] [eax] [ecx] modify [esi edi ecx]

void DWordsMover(void* dst, void* src, size_t sz);
```

В версии 11.0 точно имеется **asm{}**.

BC не хочет понимать метки в ассемблерной вставке — компилятор сказал, что не определена эта самая метка. Пришлось определить метку за пределами ASM-блока. Может быть есть более корректное решение?

Загляни в исходники RTL от BC++ 3.1 и увидишь там нечто красивое, например:

```
#define I asm
//........
I   or si,si
I   jz m1
```

```
I    mov dx,1
m1:
I    int 21h
```

и т.д.

Другой способ — компилировать с ключом **-B**. Правда, при этом могут возникнуть другие проблемы: если присутствуют имена **read** и **_read** (например), то компилятор в них запутается.

Было замечено, что борланд (3.1, например) иногда генерит разный код в зависимости от ключа **-B**. Как правило, при его наличии он становится «осторожнее» — начинает понимать, что не он один использует регистры.

Возврат адреса/ссылки локальных переменных — почему возвращает фигню:
char* myview() { char s[SIZE]; ... return s; }

Нужно поставить **static char s[SIZE];** чтобы возвращался адрес всегда существующей (статической) переменной, а автоматические переменные исчезают при выходе из функции — освобождается и может быть замусорено место из-под этих переменных на стеке.

Какие ограничения на имена с подчёркиванием?

При использовании зарезервированных имён (то есть с подчёркиваниями) возможен самый разный **undefined behavior**. Например, в компиляторе все слова с двойным подчёркиванием могут использоваться для управления файловой системой. Именно поэтому вполне допустимо (по стандарту), если Борланд в своём компиляторе, например, при встрече «нестандартной» лексемы **__asm** из сорца для VC++ просто потрёт какой-нибудь файл. На практике такого рода вариации **undefined behavior** встретить сложно, но вполне возможно.

Другие вариации **undefined behavior** — это всякие глюки при работе программы. То есть, если мы, например, в **printf** задействуем неизвестный библиотеке (нестандартный) флаг, то по стандарту вполне допустимо не проверять в библиотеке подобную фигню и передать управление куда-нибудь в область данных (ну нет в таблице переходов такого флага!).

Поведение переменных, описанных в заголовке цикла for

Переменная, объявленная в заголовке цикла (и прочих операторов!) действительна только внутри этого цикла.

Тем, кто хочет заставить вести себя также свои старые компилеры, это можно сделать следующим способом через **define**:

В новых редакциях C++ область видимости определённой в заголовке **for** переменной ограничивают телом цикла. Следующая подстановка ограничивает область видимости и для старых редакций, в которых она распространяется за пределы цикла:

```
#define for if(0);else for
```

а также для BC++ выключим вызываемые **if(0)** предупреждения:

```
"Condition is always false"
#pragma warn -ccc
```

Что есть const после имени метода?

Это означает, что этот метод не будет менять данные класса.

Методы с модификатором **const** могут изменять данные объекта, помеченные модификатором **mutable**.

```
class Bart
{
private:
 mutable int m_iSomething;
public:
 void addToSomething( int iValue ) const
 {
  m_iSomething += iValue;
 }
 Bart()
 {
  m_iSomething = 0;
 }
};
const Bart bartObject;
bartObject.addToSomething( 8 );
```

Будет скомпилировано и выполнено.

Как инициализировать статические члены класса?

```
struct a { static int i; };
int a::i; //зачем это нужно?
int main() { a::i = 11; return 0; }
```

А вот зачем:

```
struct b { int n; b(int i) :n(i) {} };
struct a { static b i; };
```

```
b a::i(0);
int main() { printf("%i\n",a::i.n); return 0; }
```

Описание некоего типа и переменная этого типа — не одно и то же. Где вы предлагаете размещать и конструировать статические поля? Может быть, в каждом файле, который включает заголовок с описанием класса? Или где?

Как не ошибиться в размере аллокируемого блока?

Для избежания подобного можно в Си сымитировать Сиплюсный new:

```
#define tmalloc(type)    ((type*)malloc(sizeof(type)))
#define amalloc(type, size)   ((type*)malloc(sizeof(type) *
(size)))
```

Более того, в последнем **define** можно поставить **(size) + 1**, чтобы гарантированно избежать проблем с завершающим нулём в строках.

Можно сделать иначе. Поскольку присвоение от **malloc()** как правило делают на типизованную переменную, то можно прямо так и написать:

```
body = malloc(sizeof(*body));
```

теперь спокойно можно менять типы не заботясь о **malloc()**. Но это верно для Си, который не ругается на присвоение **void*** к **type*** (иначе пришлось бы кастить поинтер, и компилятор изменения типа просто не пережил бы).

Вообще в С нет смысла ставить преобразования от **void*** к указательному типу явно. Более того, этот код не переносим на C++ — в проекте стандарта C++ нет **malloc()** и **free()**, их нет даже в **hosted c++** заголовках. Проще будет:

```
#ifdef __cplusplus
# define tmalloc(type)   (new type)
# define amalloc(type, size)  (new type[size])
#else
# define tmalloc(type)   malloc(sizeof(type))
# define amalloc(type, size)  malloc(sizeof(type) * (size))
#endif
```

Что такое ссылка?

Ссылка — это псевдоним (другое имя) для объекта.

Ссылки часто используются для передачи параметра по ссылке:

```
void swap(int& i, int& j)
     {
          int tmp = i;
          i = j;
          j = tmp;
     }

     int main()
     {
          int x, y;
          // ...
          swap(x,y);
     }
```

В этом примере **i** и **j** — псевдонимы для переменных **x** и **y** функции **main**. Другими словами, **i** — это **x**. Не указатель на **x** и не копия **x**, а сам **x**. Все, что вы делаете с **i**, проделывается с **x**, и наоборот.

Вот таким образом вы как программист должны воспринимать ссылки. Теперь, рискуя дать вам неверное представление, несколько слов о том, каков механизм работы ссылок. В основе ссылки **i** на объект **x** — лежит, как правило, просто машинный адрес объекта **x**. Но когда вы пишете **i++**, компилятор генерирует код, который инкрементирует **x**. В частности, сам адрес, который компилятор использует, чтобы найти **x**, остается неизменным. Программист на C может думать об этом, как если бы использовалась передача параметра по указателю, в духе языка C, но, во-первых, **&** (взятие адреса) было бы перемещено из вызывающей функции в вызываемую, и, во-вторых, в вызываемой функции были бы убраны * (разыменование). Другими словами, программист на C может думать об **i** как о макроопределении для (***p**), где **p** — это указатель на **x** (т.е., компилятор автоматически разыменовывает подлежащий указатель: **i++** заменяется на **(*p)++**, а **i = 7** на ***p = 7**).

Важное замечание: несмотря на то что в качестве ссылки в окончательном машинном коде часто используется адрес, не думайте о ссылке просто как о забавно выглядящем указателе на объект. Ссылка — это объект. Это не указатель на объект и не копия объекта. Это сам объект.

Что происходит в результате присваивания ссылке?

Вы меняете состояние ссыльного объекта (того, на который ссылается ссылка).

Помните: ссылка — это сам объект, поэтому, изменяя ссылку, вы меняете состояние объекта, на который она ссылается. На языке производителей компиляторов ссылка — это **lvalue** (**left value** — значение, которое может появиться слева от оператора присваивания).

Что происходит, когда я возвращаю из функции ссылку?

В этом случае вызов функции может оказаться с левой стороны оператора (операции) присваивания.

На первый взгляд, такая запись может показаться странной. Например, запись **f() = 7** выглядит бессмысленной. Однако, если **a** — это объект класса **Array**, для большинства людей запись **a[i] = 7** является осмысленной, хотя **a[i]** — это всего лишь замаскированный вызов функции **Array::operator[](int)**, которая является оператором обращения по индексу для класса **Array**:

```
class Array {
    public:
        int size() const;
        float& operator[] (int index);
        // ...
    };

    int main()
    {
        Array a;
        for (int i = 0; i < a.size(); ++i)
a[i] = 7; // В этой строке вызывается Array::operator[](int)
    }
```

Как можно переустановить ссылку, чтобы она ссылалась на другой объект?

Невозможно в принципе.

Невозможно отделить ссылку от ее объекта.

В отличие от указателя, ссылка, как только она привязана к объекту, не может быть «перенаправлена» на другой объект. Ссылка сама по себе ничего не представляет, у нее нет имени, она сама — это другое имя для объекта. Взятие адреса ссылки дает

адрес объекта, на который она ссылается. Помните: ссылка — это объект, на который она ссылается.

С этой точки зрения, ссылка похожа на **const** указатель, такой как **int* const p** (в отличие от указателя на **const**, такого как **const int* p**). Несмотря на большую схожесть, не путайте ссылки с указателями — это не одно и то же.

В каких случаях мне стоит использовать ссылки, и в каких — указатели?

Используйте ссылки, когда можете, а указатели — когда это необходимо.

Ссылки обычно предпочтительней указателей, когда вам не нужно их «перенаправлять». Это обычно означает, что ссылки особенно полезны в открытой (**public**) части класса. Ссылки обычно появляются на поверхности объекта, а указатели спрятаны внутри.

Исключением является тот случай, когда параметр или возвращаемый из функции объект требует выделения «охранного» значения для особых случаев. Это обычно реализуется путем взятия/возвращения указателя, и обозначением особого случая при помощи передачи нулевого указателя (**NULL**). Ссылка же не может ссылаться на разыменованный нулевой указатель.

Примечание: программисты с опытом работы на С часто недолюбливают ссылки, из-за того что передача параметра по ссылке явно никак не обозначается в вызывающем коде. Однако с обретением некоторого опыта работы на С++, они осознают, что это одна из форм сокрытия информации, которая является скорее преимуществом, чем недостатком. Т.е., программисту следует писать код в терминах задачи, а не компьютера (programmers should write code in the language of the problem rather than the language of the machine).

Что такое встроенная функция?

Встроенная функция — это функция, код которой прямо вставляется в том месте, где она вызвана. Как и макросы, определенные через **#define**, встроенные функции улучшают производительность за счет стоимости вызова и (особенно!) за счет возможности дополнительной оптимизации («процедурная интеграция»).

Как встроенные функции могут влиять на соотношение безопасности и скорости?

В обычном C вы можете получить «инкапсулированные структуры», помещая в них указатель на **void**, и заставляя его указывать на настоящие данные, тип которых неизвестен пользователям структуры. Таким образом, пользователи не знают, как интерпретировать эти данные, а функции доступа преобразуют указатель на **void** к нужному скрытому типу. Так достигается некоторый уровень инкапсуляции.

К сожалению, этот метод идет вразрез с безопасностью типов, а также требует вызова функции для доступа к любым полям структуры (если вы позволили бы прямой доступ, то его мог бы получить кто угодно, поскольку будет известно, как интерпретировать данные, на которые указывает **void***. Такое поведение со стороны пользователя приведет к сложностям при последующем изменении структуры подлежащих данных).

Стоимость вызова функции невелика, но дает некоторую прибавку. Классы C++ позволяют встраивание функций, что дает вам безопасность инкапсуляции вместе со скоростью прямого доступа. Более того, типы параметры встраиваемых функций проверяются компилятором, что является преимуществом по сравнению с сишными **#define** макросами.

Зачем мне использовать встроенные функции? Почему не использовать просто #define макросы?

Поскольку **#define** макросы опасны.

В отличие от **#define** макросов, встроенные (**inline**) функции не подвержены известным ошибкам двойного вычисления, поскольку каждый аргумент встроенной функции вычисляется только один раз. Другими словами, вызов встроенной функции — это то же самое что и вызов обычной функции, только быстрее:

```
// Макрос, возвращающий модуль (абсолютное значение) i
   #define unsafe(i) \
( (i) >= 0 ? (i) : -(i) )
// Встроенная функция, возвращающая абсолютное значение i
   inline
   int safe(int i)
   {
   return i >= 0 ? i : -i;
   }
```

```
int f();
void userCode(int x)
{
int ans;
ans = unsafe(x++); // Ошибка! x инкрементируется дважды
ans = unsafe(f()); // Опасно! f() вызывается дважды
ans = safe(x++); // Верно! x инкрементируется один раз
ans = safe(f()); // Верно! f() вызывается один раз
}
```

Также, в отличие от макросов, типы аргументов встроенных функций проверяются, и выполняются все необходимые преобразования.

Макросы вредны для здоровья; не используйте их, если это не необходимо.

Что такое ошибка в порядке статической инициализации («static initialization order fiasco»)?

Незаметный и коварный способ убить ваш проект.

Ошибка порядка статической инициализации — это очень тонкий и часто неверно воспринимаемый аспект C++. К сожалению, подобную ошибку очень сложно отловить, поскольку она происходит до вхождения в функцию **main()**.

Представьте себе, что у вас есть два статических объекта **x** и **y**, которые находятся в двух разных исходных файлах, скажем **x.cpp** и **y.cpp**. И путь конструктор объекта y вызывает какой-либо метод объекта **x**.

Вот и все. Так просто.

Проблема в том, что у вас ровно пятидесятипроцентная возможность катастрофы. Если случится, что единица трансляции с **x.cpp** будет проинициализирована первой, то все в порядке. Если же первой будет проинициализирована единица трансляции файла **y.cpp**, тогда конструктор объекта y будет запущен до конструктора **x**, и вам крышка. Т.е., конструктор **y** вызовет метод объекта **x**, когда сам **x** еще не создан.

Примечание: ошибки статической инициализации не распространяются на базовые/встроенные типы, такие как **int** или **char***. Например, если вы создаете статическую переменную типа **float**, у вас не будет проблем с порядком инициализации.

Проблема возникает только тогда, когда у вашего статического или глобального объекта есть конструктор.

Как предотвратить ошибку в порядке статической инициализации?

Используйте «создание при первом использовании», то есть, поместите ваш статический объект в функцию.

Представьте себе, что у нас есть два класса **Fred** и **Barney**. Есть глобальный объект типа **Fred**, с именем **x**, и глобальный объект типа **Barney**, с именем **y**. Конструктор **Barney** вызывает метод **goBowling()** объекта **x**. Файл **x.cpp** содержит определение объекта **x**:

```
// File x.cpp
    #include "Fred.hpp"
    Fred x;
```

Файл **y.cpp** содержит определение объекта **y**:

```
// File y.cpp
    #include "Barney.hpp"
    Barney y;
```

Для полноты представим, что конструктор **Barney::Barney()** выглядит следующим образом:

```
// File Barney.cpp
    #include "Barney.hpp"
    Barney::Barney()
    {
      // ...
      x.goBowling();
      // ...
    }
```

Проблема случается, если **y** создается раньше, чем **x**, что происходит в 50% случаев, поскольку **x** и **y** находятся в разных исходных файлах.

Есть много решений для этой проблемы, но одно очень простое и переносимое — заменить глобальный объект **Fred x**, глобальной функцией **x()**, которая возвращает объект типа **Fred** по ссылке.

```
// File x.cpp
#include "Fred.hpp"
    Fred& x()
    {
```

```
        static Fred* ans = new Fred();
        return *ans;
      }
```

Поскольку локальные статические объекты создаются в момент, когда программа в процессе работы в первый раз проходит через точку их объявления, инструкция **new Fred()** в примере выше будет выполнена только один раз: во время первого вызова функции **x()**. Каждый последующий вызов возвратит тот же самый объект **Fred** (тот, на который указывает **ans**). И далее все случаи использования объекта **x** замените на вызовы функции **x()**:

```
// File Barney.cpp
    #include "Barney.hpp"
    Barney::Barney()
    {
      // ...
      x().goBowling();
      // ...
    }
```

Это и называется «создание при первом использовании», глобальный объект **Fred** создается при первом обращении к нему.

Отрицательным моментом этой техники является тот факт, что объект **Fred** нигде не уничтожается.

Примечание: ошибки статической инициализации не распространяются на базовые/встроенные типы, такие как **int** или **char***. Например, если вы создаете статическую переменную типа **float**, у вас не будет проблем с порядком инициализации. Проблема возникает только тогда, когда у вашего статического или глобального объекта есть конструктор.

Как бороться с ошибками порядка статической инициализации объектов — членов класса?

Предположим, у вас есть класс **X**, в котором есть статический объект **Fred**:

```
// File X.hpp
class X {
    public:
      // ...
    private:
```

```
        static Fred x_;
      };
```

Естественно, этот статический член инициализируется отдельно:

```
// File X.cpp
#include "X.hpp"
Fred X::x_;
```

Опять же естественно, объект **Fred** будет использован в одном или нескольких методах класса **X**:

```
void X::someMethod()
     {
       x_.goBowling();
     }
```

Проблема проявится, если кто-то где-то каким-либо образом вызовет этот метод, до того как объект **Fred** будет создан. Например, если кто-то создает статический объект **X** и вызывает его **someMethod()** во время статической инициализации, то ваша судьба всецело находится в руках компилятора, который либо создаст **X::x_**, до того как будет вызван **someMethod()**, либо же только после.

В любом случае, всегда можно сохранить переносимость (и это абсолютно безопасный метод), заменив статический член **X::x_** на статическую функцию-член:

```
// File X.hpp
class X {
   public:
      // ...
private:
        static Fred& x();
      };
```

Естественно, этот статический член инициализируется отдельно:

```
// File X.cpp
#include "X.hpp"
 Fred& X::x()
     {
       static Fred* ans = new Fred();
       return *ans;
     }
```

После чего вы просто меняете все **x_** на **x()**:

```
void X::someMethod()
    {
      x().goBowling();
    }
```

Если для вас крайне важна скорость работы программы и вас беспокоит необходимость дополнительного вызова функции для каждого вызова **X::someMethod()**, то вы можете сделать **static Fred&**. Как вы помните, статические локальные переменные инициализируются только один раз (при первом прохождении программы через их объявление), так что **X::x()** теперь будет вызвана только один раз: во время первого вызова **X::someMethod()**:

```
void X::someMethod()
    {
      static Fred& x = X::x();
      x.goBowling();
    }
```

Примечание: ошибки статической инициализации не распространяются на базовые/встроенные типы, такие как **int** или **char***. Например, если вы создаете статическую переменную типа **float**, у вас не будет проблем с порядком инициализации. Проблема возникает только тогда, когда у вашего статического или глобального объекта есть конструктор.

Как мне обработать ошибку, которая произошла в конструкторе?

Сгенерируйте исключение.

Что такое деструктор?

Деструктор — это исполнение последней воли объекта.

Деструкторы используются для высвобождения занятых объектом ресурсов. Например, класс **Lock** может заблокировать ресурс для эксклюзивного использования, а его деструктор этот ресурс освободить. Но самый частый случай — это когда в конструкторе используется **new**, а в деструкторе — **delete**.

Деструктор это функция «готовься к смерти». Часто слово деструктор сокращается до **dtor**.

В каком порядке вызываются деструкторы для локальных объектов?

В порядке обратном тому, в каком эти объекты создавались: первым создан — последним будет уничтожен.

В следующем примере деструктор для объекта **b** будет вызван первым, а только затем деструктор для объекта **a**:

```
void userCode()
    {
      Fred a;
      Fred b;
      // ...
    }
```

В каком порядке вызываются деструкторы для массивов объектов?

В порядке обратном созданию: первым создан — последним будет уничтожен.

В следующем примере порядок вызова деструкторов будет таким: **a[9]**, **a[8]**, ..., **a[1]**, **a[0]**:

```
void userCode()
    {
      Fred a[10];
      // ...
    }
```

Могу ли я перегрузить деструктор для своего класса?

Нет.

У каждого класса может быть только один деструктор. Для класса **Fred** он всегда будет называться **Fred::~Fred()**. В деструктор никогда не передаётся никаких параметров, и сам деструктор никогда ничего не возвращает.

Всё равно вы не смогли бы указать параметры для деструктора, потому что вы никогда на вызываете деструктор напрямую (точнее, почти никогда).

Могу ли я явно вызвать деструктор для локальной переменной?

Нет!

Деструктор всё равно будет вызван еще раз при достижении закрывающей фигурной скобки **}** конца блока, в котором была создана локальная переменная. Этот вызов гарантируется языком, и он происходит автоматически; нет способа этот вызов предотвратить. Но последствия повторного вызова деструктора для одного и того же объекта могут быть плачевными. Бах! И вы покойник...

А что если я хочу, чтобы локальная переменная «умерла» раньше закрывающей фигурной скобки? Могу ли я при крайней необходимости вызвать деструктор для локальной переменной?

Нет!

Предположим, что (желаемый) побочный эффект от вызова деструктора для локального объекта **File** заключается в закрытии файла. И предположим, что у нас есть экземпляр **f** класса **File** и мы хотим, чтобы файл **f** был закрыт раньше конца своей области видимости (т.е., раньше **}**):

```
void someCode()
    {
      File f;
// ... [Этот код выполняется при открытом f] ...
// <-- Нам нужен эффект деструктора f здесь
// ... [Этот код выполняется после закрытия f] ...
    }
```

Для этой проблемы есть простое решение. Но пока запомните только следующее: нельзя явно вызывать деструктор.

Хорошо, я не буду явно вызывать деструктор. Но как мне справиться с этой проблемой?

Просто поместите вашу локальную переменную в отдельный блок **{...}**, соответствующий необходимому времени жизни этой переменной:

```
void someCode()
    {
      {
        File f;
        // ... [В этом месте f еще открыт] ...
      }
    // ^-- деструктор f будет автоматически вызван здесь!
    // ... [В этом месте f уже будет закрыт] ...
    }
```

А что делать, если я не могу поместить переменную в отдельный блок?

В большинстве случаев вы можете воспользоваться дополнительным блоком **{...}** для ограничения времени жизни вашей переменной. Но если по какой-то причине вы не можете добавить блок, добавьте функцию-член, которая будет выполнять те же действия, что и деструктор. Но помните: вы не можете сами вызывать деструктор!

Например, в случае с классом **File**, вы можете добавить метод **close()**. Обычный деструктор будет вызывать **close()**. Обратите внимание, что метод **close()** должен будет как-то отмечать объект **File**, с тем чтобы последующие вызовы не пытались закрыть уже закрытый файл. Например, можно устанавливать переменную-член **fileHandle_** в какое-нибудь неиспользуемое значение, типа **-1**, и проверять вначале, не содержит ли **fileHandle_** значение **-1**.

```
class File {
     public:
       void close();
       ~File();
       // ...
     private:
     int fileHandle_;
     // fileHandle_ >= 0 если/только если файл открыт
     };
     File::~File()
     {
       close();
     }

     void File::close()
     {
      if (fileHandle_ >= 0) {
      // ... [Вызвать системную функцию для закрытия файла]
...
         fileHandle_ = -1;
       }
     }
```

Обратите внимание, что другим методам класса **File** тоже может понадобиться проверять, не установлен ли **fileHandle_** в **-1** (т.е., не закрыт ли файл).

Также обратите внимание, что все конструкторы, которые не открывают файл, должны устанавливать **fileHandle_** в **-1**.

А могу ли я явно вызывать деструктор для объекта, созданного при помощи new?

Скорее всего, нет.

За исключением того случая, когда вы использовали синтаксис размещения для оператора **new**, вам следует просто удалять объекты при помощи **delete**, а не вызывать явно деструктор. Предположим, что вы создали объект при помощи обычного **new**:

```
Fred* p = new Fred();
```

В таком случае деструктор **Fred::~Fred()** будет автоматически вызван, когда вы удаляете объект:

```
delete p; // Вызывает p->~Fred()
```

Вам не следует явно вызывать деструктор, поскольку этим вы не освобождаете память, выделенную для объекта **Fred**. Помните: **delete p** делает сразу две вещи: вызывает деструктор и освобождает память.

Что такое «синтаксис размещения» new («placement new») и зачем он нужен?

Есть много случаев для использования синтаксиса размещения для **new**. Самое простое — вы можете использовать синтаксис размещения для помещения объекта в определенное место в памяти. Для этого вы указываете место, передавая указатель на него в оператор **new**:

```
#include <new>       // Необходимо для использования
синтаксиса размещения
#include "Fred.h"      // Определение класса Fred

    void someCode()
    {
      char memory[sizeof(Fred)];      // #1
      void* place = memory;           // #2
      Fred* f = new(place) Fred();    // #3
      // Указатели f и place будут равны
      // ...
    }
```

В строчке #1 создаётся массив из **sizeof(Fred)** байт, размер которого достаточен для хранения объекта **Fred**. В строчке #2 создаётся указатель **place**, который указывает на первый байт массива (опытные программисты на C наверняка заметят, что можно было и не создавать этот указатель; мы это сделали лишь чтобы код был более понятным). В строчке #3 фактически происходит только вызов конструктора **Fred::Fred()**. Указатель **this**

в конструкторе **Fred** будет равен указателю **place**. Таким образом, возвращаемый указатель тоже будет равен **place**.

Совет: Не используйте синтаксис размещения **new**, за исключением тех случаев, когда вам действительно нужно, чтобы объект был размещён в определённом месте в памяти. Например, если у вас есть аппаратный таймер, отображённый на определённый участок памяти, то вам может понадобиться поместить объект **Clock** по этому адресу.

Опасно: Используя синтаксис размещения **new** вы берёте на себя всю ответственность за то, что передаваемый вами указатель указывает на достаточный для хранения объекта участок памяти с тем выравниванием (alignment), которое необходимо для вашего объекта. Ни компилятор, ни библиотека не будут проверять корректность ваших действий в этом случае. Если ваш класс **Fred** должен быть выровнен по четырёхбайтовой границе, но вы передали в **new** указатель на не выровненный участок памяти, у вас могут быть большие неприятности (если вы не знаете, что такое «выравнивание» (alignment), пожалуйста, не используйте синтаксис размещения **new**). Мы вас предупредили.

Также на вас ложится вся ответственность по уничтожению размещённого объекта. Для этого вам необходимо явно вызвать деструктор:

```
void someCode()
   {
     char memory[sizeof(Fred)];
     void* p = memory;
     Fred* f = new(p) Fred();
     f->~Fred();
    // Явный вызов деструктора для размещённого объекта
   }
```

Это практически единственный случай, когда вам нужно явно вызывать деструктор.

Когда я пишу деструктор, должен ли я явно вызывать деструкторы для объектов-членов моего класса?

Нет. Никогда не надо явно вызывать деструктор (за исключением случая с синтаксисом размещения **new**.

Деструктор класса (неявный, созданный компилятором, или явно описанный вами) автоматически вызывает деструкторы

объектов-членов класса. Эти объекты уничтожаются в порядке обратном порядку их объявления в теле класса:

```
class Member {
    public:
      ~Member();
      // ...
    };

    class Fred {
    public:
      ~Fred();
      // ...
    private:
      Member x_;
      Member y_;
      Member z_;
    };

    Fred::~Fred()
    {
      // Компилятор автоматически вызывает z_.~Member()
      // Компилятор автоматически вызывает y_.~Member()
      // Компилятор автоматически вызывает x_.~Member()
    }
```

Когда я пишу деструктор производного класса, нужно ли мне явно вызывать деструктор предка?

Нет. Никогда не надо явно вызывать деструктор (за исключением случая с синтаксисом размещения **new**).

Деструктор производного класса (неявный, созданный компилятором, или явно описанный вами) автоматически вызывает деструкторы предков. Предки уничтожаются после уничтожения объектов-членов производного класса. В случае множественного наследования непосредственные предки класса уничтожаются в порядке обратном порядку их появления в списке наследования.

```
class Member {
    public:
      ~Member();
      // ...
    };
```

```
class Base {
public:
  virtual ~Base();      // Виртуальный деструктор[20.4]
  // ...
};
class Derived : public Base {
public:
  ~Derived();
  // ...
private:
  Member x_;
};

Derived::~Derived()
{
  // Компилятор автоматически вызывает x_.~Member()
  // Компилятор автоматически вызывает Base::~Base()
}
```

Примечание: в случае виртуального наследования порядок уничтожения классов сложнее. Если вы полагаетесь на порядок уничтожения классов в случае виртуального наследования, вам понадобится больше информации, чем изложено здесь.

Расскажите все-таки о пресловутых нулевых указателях

Для каждого типа указателей существует (согласно определению языка) особое значение — «нулевой указатель», которое отлично от всех других значений и не указывает на какой-либо объект или функцию. Таким образом, ни оператор **&**, ни успешный вызов **malloc()** никогда не приведут к появлению нулевого указателя. (**malloc** возвращает нулевой указатель, когда память выделить не удается, и это типичный пример использования нулевых указателей как особых величин, имеющих несколько иной смысл «память не выделена» или «теперь ни на что не указываю».)

Нулевой указатель принципиально отличается от неинициализированного указателя. Известно, что нулевой указатель не ссылается ни на какой объект; неинициализированный указатель может ссылаться на что угодно.

В приведенном выше определении уже упоминалось, что существует нулевой указатель для каждого типа указателя, и внутренние значения нулевых указателей разных типов могут отличаться. Хотя программистам не обязательно знать внутренние значения, компилятору всегда необходима информация о типе указателя, чтобы различить нулевые указатели, когда это нужно.

Как «получить» нулевой указатель в программе?

В языке Си константа **0**, когда она распознается как указатель, преобразуется компилятором в нулевой указатель. То есть, если во время инициализации, присваивания или сравнения с одной стороны стоит переменная или выражение, имеющее тип указателя, компилятор решает, что константа **0** с другой стороны должна превратиться в нулевой указатель и генерирует нулевой указатель нужного типа.

Следовательно, следующий фрагмент абсолютно корректен:

```
char *p = 0;
if(p != 0)
```

Однако, аргумент, передаваемый функции, не обязательно будет распознан как значение указателя, и компилятор может оказаться не способным распознать голый 0 как нулевой указатель. Например, системный вызов UNIX «execl» использует в качестве параметров переменное количество указателей на аргументы, завершаемое нулевым указателем. Чтобы получить нулевой указатель при вызове функции, обычно необходимо явное приведение типов, чтобы 0 воспринимался как нулевой указатель.

```
execl("/bin/sh", "sh", "-c", "ls", (char *)0);
```

Если не делать преобразования (char *), компилятор не поймет, что необходимо передать нулевой указатель и вместо этого передаст число 0. (Заметьте, что многие руководства по UNIX неправильно объясняют этот пример.)

Когда прототипы функций находятся в области видимости, передача аргументов идет в соответствии с прототипом и большинство приведений типов может быть опущено, так как прототип указывает компилятору, что необходим указатель определенного типа, давая возможность правильно преобразовать нули в указатели. Прототипы функций не могут, однако, обеспечить правильное преобразование типов в случае, когда

функция имеет список аргументов переменной длины, так что для таких аргументов необходимы явные преобразования типов. Всегда безопаснее явные преобразования в нулевой указатель, чтобы не наткнуться на функцию с переменным числом аргументов или на функцию без прототипа, чтобы временно использовать не-ANSI компиляторы, чтобы продемонстрировать, что вы знаете, что делаете. (Кстати, самое простое правило для запоминания.)

Что такое NULL и как он определен с помощью #define?

Многим программистам не нравятся нули, беспорядочно разбросанные по программам. По этой причине макрос препроцессора **NULL** определен в **<stdio.h>** или **<stddef.h>** как значение **0**. Программист, который хочет явно различать **0** как целое и **0** как нулевой указатель может использовать **NULL** в тех местах, где необходим нулевой указатель. Это только стилистическое соглашение; препроцессор преобразует **NULL** опять в 0, который затем распознается компилятором в соответствующем контексте как нулевой указатель. В отдельных случаях при передаче параметров функции может все же потребоваться явное указание типа перед **NULL** (как и перед 0).

Как #define должен определять NULL на машинах, использующих ненулевой двоичный код для внутреннего представления нулевого указателя?

Программистам нет необходимости знать внутреннее представление(я) нулевых указателей, ведь об этом обычно заботится компилятор.

Если машина использует ненулевой код для представления нулевых указателей, на совести компилятора генерировать этот код, когда программист обозначает нулевой указатель как **"0"** или **NULL**.

Следовательно, определение **NULL** как **0** на машине, для которой нулевые указатели представляются ненулевыми значениями так же правомерно как и на любой другой, так как компилятор должен (и может) генерировать корректные значения нулевых указателей в ответ на **0**, встретившийся в соответствующем контексте.

Пусть NULL был определен следующим образом: #define NULL ((char *)0). Означает ли это, что функциям можно передавать NULL без преобразования типа?

В общем, нет. Проблема в том, что существуют компьютеры, которые используют различные внутренние представления для указателей на различные типы данных. Предложенное определение через **#define** годится, когда функция ожидает в качестве передаваемого параметра указатель на **char**, но могут возникнуть проблемы при передаче указателей на переменные других типов, а верная конструкция:

```
FILE *fp = NULL;
```

может не сработать.

Тем не менее, ANSI C допускает другое определение для **NULL**:

```
#define NULL ((void *)0)
```

Кроме помощи в работе некорректным программам (но только в случае машин, где указатели на разные типы имеют одинаковые размеры, так что помощь здесь сомнительна) это определение может выявить программы, которые неверно используют **NULL** (например, когда был необходим символ ASCII NUL).

Я использую макрос #define Nullptr(type) (type *)0, который помогает задавать тип нулевого указателя

Хотя этот трюк и популярен в определенных кругах, он стоит немного. Он не нужен при сравнении и присваивании. Он даже не экономит буквы. Его использование показывает тому, кто читает программу, что автор здорово «сечет» в нулевых указателях, и требует гораздо более аккуратной проверки определения макроса, его использования и всех остальных случаев применения указателей.

Корректно ли использовать сокращенный условный оператор if(p) для проверки того, что указатель ненулевой? А что если внутреннее представление для нулевых указателей отлично от нуля?

Когда Си требует логическое значение выражения (в инструкциях **if**, **while**, **for** и **do** и для операторов **&&**, **||**, **!** и **?**) значение **false** получается, когда выражение равно нулю, а значение **true** получается в противоположном случае. Таким образом, если написано:

```
if(expr)
```

где «**expr**» — произвольное выражение, компилятор на самом деле поступает так, как будто было написано:

```
if(expr != 0)
```

Подставляя тривиальное выражение, содержащее указатель «p» вместо «**expr**», получим:

```
if(p)
```

эквивалентно

```
if(p != 0)
```

и это случай, когда происходит сравнение, так что компилятор поймет, что неявный ноль — это нулевой указатель и будет использовать правильное значение. Здесь нет никакого подвоха, компиляторы работают именно так и генерируют в обоих случаях идентичный код. Внутреннее представление указателя не имеет значения.

Оператор логического отрицания **!** может быть описан так:

```
!expr
```

на самом деле эквивалентно

```
expr?0:1
```

Читателю предлагается в качестве упражнения показать, что

```
if(!p)
```

эквивалентно

```
if(p == 0)
```

Хотя «сокращения» типа **if(p)** совершенно корректны, кое-кто считает их использование дурным стилем.

Если «NULL» и «0» эквивалентны, то какую форму из двух использовать?

Многие программисты верят, что «NULL» должен использоваться во всех выражениях, содержащих указатели как напоминание о том, что значение должно рассматриваться как указатель. Другие же чувствуют, что путаница, окружающая «**NULL**» и «**0**», только усугубляется, если «**0**» спрятать в операторе **#define** и предпочитают использовать «**0**» вместо «**NULL**». Единственного ответа не существует. Программисты на Си должны понимать, что «**NULL**» и «**0**» взаимозаменяемы и что «**0**» без преобразования типа можно без сомнения использовать при инициализации, присваивании и сравнении. Любое использование «**NULL**» (в противоположность «**0**») должно рассматриваться как ненавязчивое напоминание, что

используется указатель; программистам не нужно ничего делать (как для своего собственного понимания, так и для компилятора) для того, чтобы отличать нулевые указатели от целого числа **0**. **NULL** нельзя использовать, когда необходим другой тип нуля.

Даже если это и будет работать, с точки зрения стиля программирования это плохо.(ANSI позволяет определить **NULL** с помощью **#define как (void *)0**. Такое определение не позволит использовать **NULL** там, где не подразумеваются указатели). Особенно не рекомендуется использовать **NULL** там, где требуется нулевой код ASCII (NUL). Если необходимо, напишите собственное определение:

```
#define NUL '\0'
```

Но не лучше ли будет использовать NULL (вместо 0) в случае, когда значение NULL изменяется, быть может, на компьютере с ненулевым внутренним представлением нулевых указателей?

Нет. Хотя символические константы часто используются вместо чисел из-за того, что числа могут измениться, в данном случае причина, по которой используется **NULL**, иная. Еще раз повторим: язык гарантирует, что **0**, встреченный там, где по контексту подразумевается указатель, будет заменен компилятором на нулевой указатель. **NULL** используется только с точки зрения лучшего стиля программирования.

Я в растерянности. Гарантируется, что NULL равен 0, а нулевой указатель нет?

Термин «**null**» или «**NULL**» может не совсем обдуманно использоваться в нескольких смыслах:

1. Нулевой указатель как абстрактное понятие языка.

2. Внутреннее (на стадии выполнения) представление нулевого указателя, которое может быть отлично от нуля и различаться для различных типов указателей. О внутреннем представлении нулевого указателя должны заботиться только создатели компилятора. Программистам на Си это представление не известно.

3. Синтаксическое соглашение для нулевых указателей, символ «0».

4. Макрос **NULL** который с помощью **#define** определен как «**0**» или «**(void *)0**».

5. Нулевой код ASCII (NUL), в котором все биты равны нулю,но который имеет мало общего с нулевым указателем, разве что названия похожи.

6. «Нулевой стринг», или, что то же самое, пустой стринг ("").

Почему так много путаницы связано с нулевыми указателями? Почему так часто возникают вопросы?

Программисты на Си традиционно хотят знать больше, чем это необходимо для программирования, о внутреннем представлении кода. Тот факт, что внутреннее представление нулевых указателей для большинства машин совпадает с их представлением в исходном тексте, т.е. нулем, способствует появлению неверных обобщений. Использование макроса (**NULL**) предполагает, что значение может впоследствии измениться, или иметь другое значение для какого-нибудь компьютера. Конструкция **if(p == 0)** может быть истолкована неверно, как преобразование перед сравнением **p** к целому типу, а не **0** к типу указателя. Наконец, часто не замечают, что термин «**null**» употребляется в разных смыслах (перечисленных выше).

Хороший способ устранить путаницу — вообразить, что язык Си имеет ключевое слово (возможно, **nil**, как в Паскале), которое обозначает нулевой указатель. Компилятор либо преобразует «**nil**» в нулевой указатель нужного типа, либо сообщает об ошибке, когда этого сделать нельзя. На самом деле, ключевое слово для нулевого указателя в Си — это не «**nil**» а «**0**». Это ключевое слово работает всегда, за исключением случая, когда компилятор воспринимает в неподходящем контексте «0» без указания типа как целое число, равное нулю, вместо того, чтобы сообщить об ошибке. Программа может не работать, если предполагалось, что «**0**» без явного указания типа — это нулевой указатель.

Я все еще в замешательстве. Мне так и не понятна возня с нулевыми указателями

Следуйте двум простым правилам:

1. Для обозначения в исходном тексте нулевого указателя, используйте «**0**» или «**NULL**».

2. Если «**0**» или «**NULL**» используются как фактические аргументы при вызове функции, приведите их к типу указателя, который ожидает вызываемая функция.

Учитывая всю эту путаницу, связанную с нулевыми указателями, не лучше ли просто потребовать, чтобы их внутреннее представление было нулевым?

Если причина только в этом, то поступать так было бы неразумно, так как это неоправданно ограничит конкретную реализацию, которая (без таких ограничений) будет естественным образом представлять нулевые указатели специальными, отличными от нуля значениями, особенно когда эти значения автоматически будут вызывать специальные аппаратные прерывания, связанные с неверным доступом.

Кроме того, что это требование даст на практике? Понимание нулевых указателей не требует знаний о том, нулевое или ненулевое их внутреннее представление. Предположение о том, что внутреннее представление нулевое, не приводит к упрощению кода (за исключением некоторых случаем сомнительного использования **calloc**. Знание того, что внутреннее представление равно нулю, не упростит вызовы функций, так как размер указателя может быть отличным от размера указателя на **int**. (Если вместо «**0**» для обозначения нулевого указателя использовать «**nil**», необходимость в нулевом внутреннем представлении нулевых указателей даже бы не возникла).

Ну а если честно, на какой-нибудь реальной машине используются ненулевые внутренние представления нулевых указателей или разные представления для указателей разных типов?

Серия Prime 50 использует сегмент 07777, смещение 0 для нулевого указателя, по крайней мере, для PL/I. Более поздние модели используют сегмент **0**, смещение **0** для нулевых указателей Си, что делает необходимыми новые инструкции, такие как TCNP (проверить нулевой указатель Си), которые вводятся для совместимости с уцелевшими скверно написанными Си программами, основанными на неверных предположениях. Старые машины Prime с адресацией слов были печально знамениты тем, что указатели на байты (**char ***) у них были большего размера, чем указатели на слова (**int ***).

Серия Eclipse MV корпорации Data General имеет три аппаратно поддерживаемых типа указателей (указатели на слово, байт и бит), два из которых — **char *** и **void *** используются компиляторами Си. Указатель **word *** используется во всех других случаях.

Некоторые центральные процессоры Honeywell-Bull используют код 06000 для внутреннего представления нулевых указателей.

Серия CDC Cyber 180 использует 48-битные указатели, состоящие из кольца (ring), сегмента и смещения. Большинство пользователей имеют в качестве нулевых указателей код 0xB00000000000.

Символическая Лисп-машина с теговой архитектурой даже не имеет общеупотребительных указателей; она использует пару <**NIL,0**> (вообще говоря, несуществующий <объект, смещение> хендл) как нулевой указатель Си.

В зависимости от модели памяти, процессоры 80*86 (PC) могут использовать либо 16-битные указатели на данные и 32-битные указатели на функции, либо, наоборот, 32-битные указатели на данные и 16-битные — на функции.

Старые модели HP 3000 используют различные схемы адресации для байтов и для слов. Указатели на **char** и на **void**, имеют, следовательно, другое представление, чем указатели на **int** (на структуры и т.п.), даже если адрес одинаков.

Что означает ошибка во время исполнения «null pointer assignment» (запись по нулевому адресу). Как мне ее отследить?

Это сообщение появляется только в системе MS-DOS и означает, что произошла запись либо с помощью неинициализированного, либо нулевого указателя в нулевую область.

Отладчик обычно позволяет установить точку останова при доступе к нулевой области. Если это сделать нельзя, вы можете скопировать около 20 байт из области 0 в другую и периодически проверять, не изменились ли эти данные.

Я слышал, что char a[] эквивалентно char *a

Ничего подобного. (То, что вы слышали, касается формальных параметров функций.) Массивы — не указатели. Объявление массива «**char a[6];**» требует определенного места для шести символов, которое будет известно под именем «**a**». То есть, существует место под именем «**a**», в которое могут быть помещены **6** символов. С другой стороны, объявление указателя «**char *p;**» требует места только для самого указателя. Указатель

будет известен под именем «**p**» и может указывать на любой символ (или непрерывный массив символов).

Важно понимать, что ссылка типа **x[3]** порождает разный код в зависимости от того, массив **x** или указатель.

В случае выражения **p[3]** компилятор генерирует код, чтобы начать с позиции «p», считывает значение указателя, прибавляет к указателю 3 и, наконец, читает символ, на который указывает указатель.

Что понимается под «эквивалентностью указателей и массивов» в Си?

Большая часть путаницы вокруг указателей в Си происходит от непонимания этого утверждения. «Эквивалентность» указателей и массивов не позволяет говорить не только об идентичности, но и о взаимозаменяемости.

«Эквивалентность» относится к следующему ключевому определению: значение типа массив **T**, которое появляется в выражении, превращается (за исключением трех случаев) в указатель на первый элемент массива; тип результирующего указателя — указатель на **T**. (Исключение составляют случаи, когда массив оказывается операндом **sizeof**, оператора **&** или инициализатором символьной строки для массива литер.)

Вследствие этого определения нет заметной разницы в поведении оператора индексирования **[]**, если его применять к массивам и указателям. Согласно правилу, приведенному выше, в выражении типа **a[i]** ссылка на массив «**a**» превращается в указатель и дальнейшая индексация происходит так, как будто существует выражение с указателем **p[i]** (хотя доступ к памяти будет различным). В любом случае выражение **x[i]**, где **x** — массив или указатель) равно по определению ***((x)+(i))**.

Почему объявления указателей и массивов взаимозаменяемы в в качестве формальных параметров?

Так как массивы немедленно превращаются в указатели, массив на самом деле не передается в функцию. По общему правилу, любое похожее на массив объявление параметра:

```
f(a)
char a[];
```

рассматривается компилятором как указатель, так что если был передан массив, функция получит:

```
f(a)
```

```
char *a;
```

Это превращение происходит только для формальных параметров функций, больше нигде. Если это превращение раздражает вас, избегайте его; многие пришли к выводу, что порождаемая этим путаница перевешивает небольшое преимущество от того, что объявления смотрятся как вызов функции и/или напоминают о том, как параметр будет использоваться внутри функции.

Как массив может быть значением типа lvalue, если нельзя присвоить ему значение?

Стандарт ANSI C определяет «модифицируемое lvalue», но массив к этому не относится.

Почему sizeof неправильно определяет размер массива, который передан функции в качестве параметра?

Оператор **sizeof** сообщает размер указателя, который на самом деле получает функция.

Кто-то объяснил мне, что массивы это на самом деле только постоянные указатели

Это слишком большое упрощение. Имя массива — это константа, следовательно, ему нельзя присвоить значение, но массив — это не указатель.

С практической точки зрения в чем разница между массивами и указателями?

Массивы автоматически резервируют память, но не могут изменить расположение в памяти и размер. Указатель должен быть задан так, чтобы явно указывать на выбранный участок памяти (возможно с помощью **malloc**), но он может быть по нашему желанию переопределен (т.е. будет указывать на другие объекты) и, кроме того, указатель имеет много других применений, кроме службы в качестве базового адреса блоков памяти.

В рамках так называемой эквивалентности массивов и указателей, массивы и указатели часто оказываются взаимозаменяемыми.

Особенно это касается блока памяти, выделенного функцией **malloc**, указатель на который часто используется как настоящий массив.

Я наткнулся на шуточный код, содержащий «выражение» 5["abcdef"]. Почему такие выражения возможны в Си?

Да, индекс и имя массива можно переставлять в Си. Этот забавный факт следует из определения индексации через указатель, а именно, **a[e]** идентично ***((a)+(e))**, для любого выражения **e** и основного выражения **a**, до тех пор пока одно из них будет указателем, а другое целочисленным выражением. Это неожиданная коммутативность часто со странной гордостью упоминается в С-текстах, но за пределами Соревнований по Непонятному Программированию (Obfuscated C Contest)

Мой компилятор ругается, когда я передаю двумерный массив функции, ожидающей указатель на указатель

Правило, по которому массивы превращаются в указатели не может применяться рекурсивно. Массив массивов (т.е. двумерный массив в Си) превращается в указатель на массив, а не в указатель на указатель.

Указатели на массивы могут вводить в заблуждение и применять их нужно с осторожностью. (Путаница еще более усугубляется тем, что существуют некорректные компиляторы, включая некоторые версии pcc и полученные на основе **pcc** программы **lint**, которые неверно воспринимают присваивание многоуровневым указателям многомерных массивов.) Если вы передаете двумерный массив функции:

```
int array[NROWS][NCOLUMNS];
f(array);
```

описание функции должно соответствовать

```
 f(int a[][NCOLUMNS]) {...}
```

или

```
f(int (*ap)[NCOLUMNS]) {...}   /* ap - указатель на массив
*/
```

В случае, когда используется первое описание, компилятор неявно осуществляет обычное преобразование «массива массивов» в «указатель на массив»; во втором случае указатель на массив задается явно.

Так как вызываемая функция не выделяет место для массива, нет необходимости знать его размер, так что количество «строк» **NROWS** может быть опущено. «Форма» массива по-прежнему важна, так что размер «столбца» **NCOLUMNS**

должен быть включен (а для массивов размерности 3 и больше, все промежуточные размеры).

Если формальный параметр функции описан как указатель на указатель, то передача функции в качестве параметра двумерного массива будет, видимо, некорректной.

Как писать функции, принимающие в качестве параметра двумерные массивы, «ширина» которых во время компиляции неизвестна?

Это непросто. Один из путей — передать указатель на элемент **[0][0]** вместе с размерами и затем симулировать индексацию «вручную»:

```
f2(aryp, nrows, ncolumns)
int *aryp;
int nrows, ncolumns;
{ ... array[i][j] это aryp[i * ncolumns + j] ... }
```

Этой функции массив может быть передан так:

```
f2(&array[0][0], NROWS, NCOLUMNS);
```

Нужно, однако, заметить, что программа,выполняющая индексирование многомерного массива «вручную» не полностью соответствует стандарту ANSI C; поведение **(&array[0][0])[x]** не определено при **x > NCOLUMNS**.

gcc разрешает объявлять локальные массивы, которые имеют размеры, задаваемые аргументами функции, но это — нестандартное расширение.

Как объявить указатель на массив?

Обычно этого делать не нужно. Когда случайно говорят об указателе на массив, обычно имеют в виду указатель на первый элемент массива.

Вместо указателя на массив рассмотрим использование указателя на один из элементов массива. Массивы типа **T** превращаются в указатели типа **T**, что удобно; индексация или увеличение указателя позволяет иметь доступ к отдельным элементам массива. Истинные указатели на массивы при увеличении или индексации указывают на следующий массив и в общем случае если и полезны, то лишь при операциях с массивами массивов.

Если действительно нужно объявить указатель на целый массив, используйте что-то вроде **int (*ap)[N];** где **N** — размер массива. Если размер массива неизвестен, параметр **N** может быть

опущен, но получившийся в результате тип «указатель на массив неизвестного размера» — бесполезен.

Исходя из того, что ссылки на массив превращаются в указатели, скажите в чем разница для массива int array[NROWS][NCOLUMNS]; между array и &array?

Согласно ANSI/ISO стандарту Си, **&array** дает указатель типа «**указатель-на-массив-Т**», на весь массив.

В языке Си до выхода стандарта ANSI оператор **&** в **&array** игнорировался, порождая предупреждение компилятора. Все компиляторы Си, встречая просто имя массива, порождают указатель типа **указатель-на-Т**, т.е. на первый элемент массива.

Как динамически выделить память для многомерного массива?

Лучше всего выделить память для массива указателей, а затем инициализировать каждый указатель так, чтобы он указывал на динамически создаваемую строку. Вот пример для двумерного массива:

```
int **array1 = (int **)malloc(nrows * sizeof(int *));
for(i = 0; i < nrows; i++)
array1[i] = (int *)malloc(ncolumns * sizeof(int));
```

(В «реальной» программе, **malloc** должна быть правильно объявлена, а каждое возвращаемое **malloc** значение — проверено.)

Можно поддерживать монолитность массива, (одновременно затрудняя последующий перенос в другое место памяти отдельных строк), с помощью явно заданных арифметических действий с указателями:

```
int **array2 = (int **)malloc(nrows * sizeof(int *));
   array2[0] = (int *)malloc(nrows * ncolumns *
sizeof(int));
          for(i = 1; i < nrows; i++)
          array2[i] = array2[0] + i * ncolumns;
```

В любом случае доступ к элементам динамически задаваемого массива может быть произведен с помощью обычной индексации: **array[i][j]**.

Если двойная косвенная адресация, присутствующая в приведенных выше примерах, вас по каким-то причинам не устраивает, можно имитировать двумерный массив с помощью динамически задаваемого одномерного массива:

```
int *array3 = (int *)malloc(nrows * ncolumns * sizeof(int));
```

Теперь, однако, операции индексирования нужно выполнять вручную, осуществляя доступ к элементу **i,j** с помощью **array3[i*ncolumns+j]**. (Реальные вычисления можно скрыть в макросе, однако вызов макроса требует круглых скобок и запятых, которые не выглядят в точности так, как индексы многомерного массива.)

Наконец, можно использовать указатели на массивы:

```
int (*array4)[NCOLUMNS] =
 (int(*)[NCOLUMNS])malloc(nrows * sizeof(*array4));,
```

но синтаксис становится устрашающим, и «всего лишь» одно измерение должно быть известно во время компиляции.

Пользуясь описанными приемами, необходимо освобождать память, занимаемую массивами (это может проходить в несколько шагов), когда они больше не нужны, и не следует смешивать динамически создаваемые массивы с обычными, статическими.

Как мне равноправно использовать статически и динамически задаваемые многомерные массивы при передаче их в качестве параметров функциям?

Идеального решения не существует. Возьмем объявления

```
int array[NROWS][NCOLUMNS];
int **array1;
int **array2;
int *array3;
int (*array4)[NCOLUMNS];
```

соответствующие способам выделения памяти и функции, объявленные как:

```
f1(int a[][NCOLUMNS], int m, int n);
f2(int *aryp, int nrows, int ncolumns);
f3(int **pp, int m, int n);
```

Тогда следующие вызовы должны работать так, как ожидается:

```
f1(array, NROWS, NCOLUMNS);
f1(array4, nrows, NCOLUMNS);
f2(&array[0][0], NROWS, NCOLUMNS);
f2(*array2, nrows, ncolumns);
f2(array3, nrows, ncolumns);
f2(*array4, nrows, NCOLUMNS);
f3(array1, nrows, ncolumns);
```

```
f3(array2, nrows, ncolumns);
```

Следующие два вызова, возможно, будут работать, но они включают сомнительные приведения типов, и работают лишь в том случае, когда динамически задаваемое число столбцов **ncolumns** совпадает с **NCOLUMS**:

```
f1((int (*)[NCOLUMNS])(*array2), nrows, ncolumns);
f1((int (*)[NCOLUMNS])array3, nrows, ncolumns);
```

Необходимо еще раз отметить, что передача **&array[0][0]** функции **f2** не совсем соответствует стандарту.

Если вы способны понять, почему все вышеперечисленные вызовы работают и написаны именно так, а не иначе, и если вы понимаете, почему сочетания, не попавшие в список, работать не будут, то у вас очень хорошее понимание массивов и указателей (и нескольких других областей) Си.

Вот изящный трюк: если я пишу int realarray[10]; int *array = &realarray[-1];, то теперь можно рассматривать «array» как массив, у которого индекс первого элемента равен единице

Хотя этот прием внешне привлекателен, он не удовлетворяет стандартам Си. Арифметические действия над указателями определены лишь тогда, когда указатель ссылается на выделенный блок памяти или на воображаемый завершающий элемент, следующий сразу за блоком. В противном случае поведение программы не определено, даже если указатель не переназначается. Код, приведенный выше, плох тем, что при уменьшении смещения может быть получен неверный адрес (возможно, из-за циклического перехода адреса при пересечении границы сегмента).

У меня определен указатель на char, который указывает еще и на int, причем мне необходимо переходить к следующему элементу типа int. Почему ((int *)p)++; не работает?

В языке Си оператор преобразования типа не означает «будем действовать так, как будто эти биты имеют другой тип»; это оператор, который действительно выполняет преобразования, причем по определению получается значение типа **rvalue**, которому нельзя присвоить новое значение и к которому не применим оператор ++. (Следует считать аномалией то, что компиляторы **pcc** и расширения **gcc** вообще воспринимают выражения приведенного выше типа.)

Скажите то, что думаете:

```
p = (char *)((int *)p + 1);
```

или просто

```
p += sizeof(int);
```

Могу я использовать void **, чтобы передать функции по ссылке обобщенный указатель?

Стандартного решения не существует, поскольку в Си нет общего типа **указатель-на-указатель**. **void *** выступает в роли обобщенного указателя только потому, что автоматически осуществляются преобразования в ту и другую сторону, когда встречаются разные типы указателей. Эти преобразования не могут быть выполнены (истинный тип указателя неизвестен), если осуществляется попытка косвенной адресации, когда **void **** указывает на что-то отличное от **void ***.

Почему не работает фрагмент кода:
char *answer;
printf("Type something:\n");
gets(answer);
printf("You typed \"%s\"\n", answer);

Указатель «**answer**», который передается функции **gets** как место, в котором должны храниться вводимые символы, не инициализирован, т.е. не указывает на какое-то выделенное место. Иными словами, нельзя сказать, на что указывает «**answer**». (Так как локальные переменные не инициализируются, они вначале обычно содержат «мусор», то есть даже не гарантируется, что в начале «**answer**» — это нулевой указатель.

Простейший способ исправить программу — использовать локальный массив вместо указателя, предоставив компилятору заботу о выделении памяти:

```
#include <string.h>
char answer[100], *p;
printf("Type something:\n");
  fgets(answer, sizeof(answer), stdin);
  if((p = strchr(answer, '\n')) != NULL)
  *p = '\0';
  printf("You typed \"%s\"\n", answer);
```

Заметьте, что в этом примере используется **fgets()** вместо **gets()**, что позволяет указать размер массива, так что выход за пределы массива, когда пользователь введет слишком длинную

строку, становится невозможным. (К сожалению, **fgets()** не удаляет автоматически завершающий символ конца строки **\n**, как это делает **gets()**). Для выделения памяти можно также использовать **malloc()**.

Не могу заставить работать strcat. В моей программе
char *s1 = "Hello, ";
char *s2 = "world!";
char *s3 = strcat(s1, s2);
но результаты весьма странные

Проблема снова состоит в том, что не выделено место для результата объединения. Си не поддерживает автоматически переменные типа **string**.

Компиляторы Си выделяют память только под объекты, явно указанные в исходном тексте (в случае стрингов это может быть массив литер или символы, заключенные в двойные кавычки). Программист должен сам позаботиться о том, чтобы была выделена память для результата, который получается в процессе выполнения программы, например результата объединения строк. Обычно это достигается объявлением массива или вызовом **malloc**.

Функция **strcat** не выделяет память; вторая строка присоединяется к первой. Следовательно, одно из исправлений — в задании первой строки в виде массива достаточной длины:

```
char s1[20] = "Hello, ";
```

Так как **strcat** возвращает указатель на первую строку (в нашем случае **s1**), переменная **s3** — лишняя.

В справочнике о функции strcat сказано, что она использует в качестве аргументов два указателя на char. Откуда мне знать о выделении памяти?

Как правило, при использовании указателей всегда необходимо иметь в виду выделение памяти, по крайней мере, быть уверенным, что компилятор делает это для вас. Если в документации на библиотечную функцию явно ничего не сказано о выделении памяти, то обычно это проблема вызывающей функции.

Краткое описание функции в верхней части страницы справочника в стиле UNIX может ввести в заблуждение. Приведенные там фрагменты кода ближе к определению, необходимому для разработчика функции, чем для того, кто будет

эту функцию вызывать. В частности, многие функции, имеющие в качестве параметров указатели (на структуры или стринги, например), обычно вызываются с параметрами, равными адресам каких-то уже существующих объектов (структур или массивов). Другой распространенный пример — функция **stat()**.

Предполагается, что функция, которую я использую, возвращает строку, но после возврата в вызывающую функцию эта строка содержит «мусор»

Убедитесь, что правильно выделена область памяти, указатель на которую возвращает ваша функция. Функция должна возвращать указатель на статически выделенную область памяти или на буфер, передаваемый функции в качестве параметра, или на память, выделенную с помощью **malloc()**, но не на локальный (auto) массив. Другими словами, никогда не делайте ничего похожего на:

```
char *f()
{
   char buf[10];
   /* ... */
   return buf;
}
```

Приведем одну поправку (непригодную в случае, когда **f()** вызывается рекурсивно, или когда одновременно нужны несколько возвращаемых значений):

```
static char buf[10];
```

Почему в некоторых исходных текстах значения, возвращаемые malloc(), аккуратно преобразуются в указатели на выделяемый тип памяти?

До того, как стандарт ANSI/ISO ввел обобщенный тип указателя **void ***, эти преобразования были обычно необходимы для подавления предупреждений компилятора о приравнивании указателей разных типов. (Согласно стандарту C ANSI/ISO, такие преобразования типов указателей не требуются).

Можно использовать содержимое динамически выделяемой памяти после того как она освобождена?

Нет. Иногда в старых описаниях **malloc()** говорилось, что содержимое освобожденной памяти «остается неизменным»; такого рода поспешная гарантия никогда не была универсальной и не требуется стандартом ANSI.

Немногие программисты стали бы нарочно использовать содержимое освобожденной памяти, но это легко сделать

нечаянно. Рассмотрите следующий (корректный) фрагмент программы, в котором освобождается память, занятая односвязным списком:

```
struct list *listp, *nextp;
for(listp = base; listp != NULL; listp = nextp) {
nextp = listp->next;
free((char *)listp);
}
```

и подумайте, что получится, если будет использовано на первый взгляд более очевидное выражение для тела цикла:

```
listp = listp->next
```

без временного указателя **nextp**.

Откуда free() знает, сколько байт освобождать?

Функции **malloc/free** запоминают размер каждого выделяемого и возвращаемого блока, так что не нужно напоминать размер освобождаемого блока.

А могу я узнать действительный размер выделяемого блока?

Нет универсального ответа.

Я выделяю память для структур, которые содержат указатели на другие динамически создаваемые объекты. Когда я освобождаю память, занятую структурой, должен ли я сначала освободить память, занятую подчиненным объектом?

Да. В общем, необходимо сделать так, чтобы каждый указатель, возвращаемый **malloc()** был передан **free()** точно один раз (если память освобождается).

В моей программе сначала с помощью malloc() выделяется память, а затем большое количество памяти освобождается с помощью free(), но количество занятой памяти (так сообщает команда операционной системы) не уменьшается

Большинство реализаций **malloc/free** не возвращают освобожденную память операционной системе (если таковая имеется), а просто делают освобожденную память доступной для будущих вызовов **malloc()** в рамках того же процесса.

Должен ли я освобождать выделенную память перед возвратом в операционную систему?

Делать это не обязательно. Настоящая операционная система восстанавливает состояние памяти по окончании работы программы.

Тем не менее, о некоторых персональных компьютерах известно, что они ненадежны при восстановлении памяти, а из стандарта ANSI/ISO можно лишь получить указание, что эти вопросы относятся к «качеству реализации».

Правильно ли использовать нулевой указатель в качестве первого аргумента функции realloc()? Зачем это нужно?

Это разрешено стандартом ANSI C (можно также использовать **realloc(...,0)** для освобождения памяти), но некоторые ранние реализации Си это не поддерживают, и мобильность в этом случае не гарантируется. Передача нулевого указателя **realloc()** может упростить написание самостартующего алгоритма пошагового выделения памяти.

В чем разница между calloc и malloc? Получатся ли в результате применения calloc корректные значения нулевых указателей и чисел с плавающей точкой? Освобождает ли free память, выделенную calloc, или нужно использовать cfree?

По существу **calloc(m,n)** эквивалентна:

```
p = malloc(m * n);
memset(p, 0, m * n);
```

Заполнение нулями означает зануление всех битов, и, следовательно, не гарантирует нулевых значений для указателей и для чисел с плавающей точкой. Функция **free** может (и должна) использоваться для освобождения памяти, выделенной **calloc**.

Что такое alloca и почему использование этой функции обескураживает?

alloca выделяет память, которая автоматически освобождается, когда происходит возврат из функции, в которой вызывалась **alloca**. То есть, память, выделенная **alloca**, локальна по отношению к «стековому кадру» или контексту данной функции.

Использование **alloca** не может быть мобильным, реализации этой функции трудны на машинах без стека. Использование этой функции проблематично (и очевидная реализация на машинах со стеком не удается), когда возвращаемое ей значение непосредственно передается другой функции, как, например, в **fgets(alloca(100), 100, stdin)**.

По изложенным выше причинам **alloca** (вне зависимости от того, насколько это может быть полезно) нельзя использовать в программах, которые должны быть в высокой степени мобильны.

Почему вот такой код: a[i] = i++; не работает?

Подвыражение **i++** приводит к побочному эффекту — значение **i** изменяется, что приводит к неопределенности, если **i** уже встречается в том же выражении.

Пропустив код
int i = 7;
printf("%d\n", i++ * i++);
через свой компилятор, я получил на выходе 49. А разве, независимо от порядка вычислений, результат не должен быть равен 56?

Хотя при использовании постфиксной формы операторов **++** и **--** увеличение и уменьшение выполняется после того как первоначальное значение использовано, тайный смысл слова «после» часто понимается неверно. Не гарантируется, что увеличение или уменьшение будет выполнено немедленно после использования первоначального значения перед тем как будет вычислена любая другая часть выражения. Просто гарантируется, что изменение будет произведено в какой-то момент до окончания вычисления (перед следующей «точкой последовательности» в терминах ANSI C). В приведенном примере компилятор умножил предыдущее значение само на себя и затем дважды увеличил **i** на **1**.

Поведение кода, содержащего многочисленные двусмысленные побочные эффекты не определено. Даже не пытайтесь выяснить, как ваш компилятор все это делает (в противоположность неумным упражнениям во многих книгах по C).

Я экспериментировал с кодом:
int i = 2;
i = i++;
Некоторые компиляторы выдавали i=2, некоторые 3, но один выдал 4. Я знаю, что поведение не определено, но как можно получить 4?

Неопределенное (undefined) поведение означает, что может случиться все что угодно.

Люди твердят, что поведение не определено, а я попробовал ANSI-компилятор и получил то, что ожидал

Компилятор делает все, что ему заблагорассудится, когда встречается с неопределенным поведением (до некоторой степени это относится и к случаю зависимого от реализации и неописанного поведения). В частности, он может делать то, что вы ожидаете. Неблагоразумно, однако, полагаться на это.

Могу я использовать круглые скобки, чтобы обеспечить нужный мне порядок вычислений? Если нет, то разве приоритет операторов не обеспечивает этого?

Круглые скобки, как и приоритет операторов обеспечивают лишь частичный порядок при вычислении выражений. Рассмотрим выражение:

```
f() + g() * h()    .
```

Хотя известно, что умножение будет выполнено раньше сложения, нельзя ничего сказать о том, какая из трех функций будет вызвана первой.

Тогда как насчет операторов &&, || и запятой? Я имею в виду код типа if((c = getchar()) == EOF || c == '\n')"

Для этих операторов, как и для оператора **?:** существует специальное исключение; каждый из них подразумевает определенный порядок вычислений, т.е. гарантируется вычисление слева-направо.

Если я не использую значение выражения, то как я должен увеличивать переменную i: так: ++i или так: i++?

Применение той или иной формы сказывается только на значении выражения, обе формы полностью эквивалентны, когда требуются только их побочные эффекты.

Почему неправильно работает код:
int a = 1000, b = 1000;
long int c = a * b;

Согласно общим правилам преобразования типов языка Си, умножение выполняется с использованием целочисленной арифметики, и результат может привести к переполнению и/или усечен до того как будет присвоен стоящей слева переменной типа **long int**. Используйте явное приведение типов, чтобы включить арифметику длинных целых:

```
long int c = (long int)a * b;
```

Заметьте, что код **(long int)(a * b)** не приведет к желаемому результату.

Что такое стандарт ANSI C?

В 1983 году Американский институт национальных стандартов (ANSI) учредил комитет X3J11, чтобы разработать стандарт языка Си. После длительной и трудной работы, включающей выпуск нескольких публичных отчетов, работа

комитета завершилась 14 декабря 1989 г.созданием стандарта ANS X3.159-1989. Стандарт был опубликован весной 1990 г.

В большинстве случаев ANSI C узаконил уже существующую практику и сделал несколько заимствований из C++ (наиболее важное — введение прототипов функций). Была также добавлена поддержка национальных наборов символов (включая подвергшиеся наибольшим нападкам трехзнаковые последовательности). Стандарт ANSI C формализовал также стандартную библиотеку.

Опубликованный стандарт включает «Комментарии» («Rationale»), в которых объясняются многие решения и обсуждаются многие тонкие вопросы, включая несколько затронутых здесь. («Комментарии» не входят в стандарт ANSI X3.159-1989, они приводятся в качестве дополнительной информации.)

Стандарт ANSI был принят в качестве международного стандарта ISO/IEC 9899:1990, хотя нумерация разделов иная (разделы 2 — 4 стандарта ANSI соответствуют разделам 5 — 7 стандарта ISO), раздел «Комментарии» не был включен.

Как получить копию Стандарта?

ANSI X3.159 был официально заменен стандартом ISO 9899. Копию стандарта можно получить по адресу:

```
American National Standards Institute
11 W. 42nd St., 13th floor
New York, NY 10036 USA
(+1) 212 642 4900
```

или

```
Global Engineering Documents
2805 McGaw Avenue
Irvine, CA 92714 USA
(+1) 714 261 1455
(800) 854 7179  (U.S. & Canada)
```

В других странах свяжитесь с местным комитетом по стандартам или обратитесь в Национальный Комитет по Стандартам в Женеве:

```
ISO Sales
Case Postale 56
CH-1211 Geneve 20
```

```
Switzerland
```

Есть ли у кого-нибудь утилиты для перевода С-программ, написанных в старом стиле, в ANSI C и наоборот? Существуют ли программы для автоматического создания прототипов?

Две программы, **protoize** и **unprotoize** осуществляют преобразование в обе стороны между функциями, записанными в новом стиле с прототипами, и функциями, записанными в старом стиле. (Эти программы не поддерживают полный перевод между «классическим» и ANSI C).

Упомянутые программы были сначала вставками в FSF GNU компилятор C, **gcc**, но теперь они — часть дистрибутива **gcc**.

Программа **unproto** — это фильтр, располагающийся между препроцессором и следующим проходом компилятора — на лету переводит большинство особенностей ANSI C в традиционный Си.

GNU пакет GhostScript содержит маленькую программу **ansi2knr**.

Есть несколько генераторов прототипов, многие из них — модификации программы **lint**. Версия 3 программы CPROTO была помещена в конференцию comp.sources.misc в марте 1992 г. Есть другая программа, которая называется «ctxtract».

В заключение хочется спросить: так ли уж нужно преобразовывать огромное количество старых программ в ANSI C? Старый стиль написания функций все еще допустим.

Я пытаюсь использовать ANSI-строкообразующий оператор #, чтобы вставить в сообщение значение символической константы, но вставляется формальный параметр макроса, а не его значение

Необходимо использовать двухшаговую процедуру для того чтобы макрос раскрывался как при строкообразовании:

```
#define str(x) #x
#define xstr(x) str(x)
#define OP plus
char *opname = xstr(OP);
```

Такая процедура устанавливает **opname** равным «**plus**», а не «**OP**».

Такие же обходные маневры необходимы при использовании оператора склеивания лексем ##, когда нужно соединить значения (а не имена формальных параметров) двух макросов.

Не понимаю, почему нельзя использовать неизменяемые значения при инициализации переменных и задании размеров массивов, как в следующем примере:

```
const int n = 5;
int a[n];
```

Квалификатор **const** означает «только для чтения». Любой объект, квалифицированный как **const**, представляет собой нормальный объект, существующий во время исполнения программы, которому нельзя присвоить другое значение. Следовательно, значение такого объекта — это не константное выражение в полном смысле этого слова. (В этом смысле Си не похож на C++). Если есть необходимость в истинных константах, работающих во время компиляции, используйте препроцессорную директиву **#define**.

Какая разница между «char const *p» и «char * const p»?

«**char const *p**» — это указатель на постоянную литеру (ее нельзя изменить); «**char * const p**» — это неизменяемый указатель на переменную (ее можно менять) типа **char**. Зарубите это себе на носу.

Почему нельзя передать char ** функции, ожидающей const char **?

Можно использовать **указатель-на-Т** любых типов **Т**, когда ожидается **указатель-на-const-Т**, но правило (точно определенное исключение из него), разрешающее незначительные отличия в указателях, не может применяться рекурсивно, а только на самом верхнем уровне.

Необходимо использовать точное приведение типов (т.е. в данном случае (**const char ****)) при присвоении или передаче указателей, которые имеют различия на уровне косвенной адресации, отличном от первого.

Мой ANSI компилятор отмечает несовпадение, когда встречается с декларациями:

```
extern int func(float);
  int func(x)
  float     x;
  {...
```

Вы смешали декларацию в новом стиле «**extern int func(float);**» с определением функции в старом стиле «**int func(x) float x;**».

Смешение стилей, как правило, безопасно, но только не в этом случае. Старый Си (и ANSI C при отсутствии прототипов и в

списках аргументов переменной длины) «расширяет» аргументы определенных типов при передаче их функциям. Аргументы типа **float** преобразуются в тип **double**, литеры и короткие целые преобразуются в тип **int**. (Если функция определена в старом стиле, параметры автоматически преобразуются в теле функции к менее емким, если таково их описание там.)

Это затруднение может быть преодолено либо с помощью определений в новом стиле:

```
int func(float x) { ... }
```

либо с помощью изменения прототипа в новом стиле таким образом, чтобы он соответствовал определению в старом стиле:

```
extern int func(double);
```

(В этом случае для большей ясности было бы желательно изменить и определение в старом стиле так, чтобы параметр, если только не используется его адрес, был типа **double**.)

Возможно, будет безопасней избегать типов **char**, **short int**, **float** для возвращаемых значений и аргументов функций.

Можно ли смешивать определения функций в старом и новом стиле?

Смешение стилей абсолютно законно, если соблюдается осторожность. Заметьте, однако, что определение функций в старом стиле считается выходящим из употребления, и в один прекрасный момент поддержка старого стиля может быть прекращена.

Почему объявление extern f(struct x {int s;} *p); порождает невнятное предупреждение «struct x introduced in prototype scope» (структура объявлена в зоне видимости прототипа)?

В странном противоречии с обычными правилами для областей видимости структура, объявленная только в прототипе, не может быть совместима с другими структурами, объявленными в этом же файле. Более того, вопреки ожиданиям тег структуры не может быть использован после такого объявления (зона видимости объявления простирается до конца прототипа). Для решения проблемы необходимо, чтобы прототипу предшествовало «пустое» объявление:

```
struct x;
```

которое зарезервирует место в области видимости файла для определения структуры **x**. Определение будет завершено объявлением структуры внутри прототипа.

У меня возникают странные сообщения об ошибках внутри кода, «выключенного» с помощью #ifdef

Согласно ANSI C, текст, «выключенный» с помощью **#if**, **#ifdef** или **#ifndef** должен состоять из «корректных единиц препроцессирования».

Это значит, что не должно быть незакрытых комментариев или кавычек (обратите особое внимание, что апостроф внутри сокращенно записанного слова смотрится как начало литерной константы).

Внутри кавычек не должно быть символов новой строки. Следовательно, комментарии и псевдокод всегда должны находиться между непосредственно предназначенными для этого символами начала и конца комментария /* и */.

Могу я объявить main как void, чтобы прекратились раздражающие сообщения «main return no value»? (Я вызываю exit(), так что main ничего не возвращает)

Нет. **main** должна быть объявлена как возвращающая **int** и использующая либо два, либо ни одного аргумента (подходящего типа). Если используется **exit()**, но предупреждающие сообщения не исчезают, вам нужно будет вставить лишний **return**, или использовать, если это возможно, директивы вроде «**notreached**».

Объявление функции как **void** просто не влияет на предупреждения компилятора; кроме того, это может породить другую последовательность вызова/возврата, несовместимую с тем, что ожидает вызывающая функция (в случае **main** это исполняющая система языка Си).

В точности ли эквивалентен возврат статуса с помощью exit(status) возврату с помощью return?

Формально, да, хотя несоответствия возникают в некоторых старых нестандартных системах, в тех случаях, когда данные, локальные для **main()**, могут потребоваться в процессе завершения выполнения (может быть при вызовах **setbuf()** или **atexit()**), или при рекурсивном вызове **main()**.

Почему стандарт ANSI гарантирует только шесть значимых символов (при отсутствии различия между прописными и строчными символами) для внешних идентификаторов?

Проблема в старых компоновщиках, которые не зависят ни от стандарта ANSI, ни от разработчиков компиляторов. Ограничение состоит в том, что только первые шесть символов

значимы, а не в том, что длина идентификатора ограничена шестью символами. Это ограничение раздражает, но его нельзя считать невыносимым. В Стандарте оно помечено как «выходящее из употребления», так что в следующих редакциях оно, вероятно, будет ослаблено.

Эту уступку современным компоновщикам, ограничивающим количество значимых символов, обязательно нужно делать, не обращая внимания на бурные протесты некоторых программистов. (В «Комментариях» сказано, что сохранение этого ограничения было «наиболее болезненным».

Если вы не согласны или надеетесь с помощью какого-то трюка заставить компилятор, обремененный ограничивающим количество значимых символов компоновщиком, понимать большее количество этих символов, читайте превосходно написанный раздел 3.1.2 X3.159 «Комментариев».

Какая разница между memcpy и memmove?

memmove гарантирует правильность операции копирования, если две области памяти перекрываются. memcpy не дает такой гарантии и, следовательно, может быть более эффективно реализована. В случае сомнений лучше применять **memmove**.

Мой компилятор не транслирует простейшие тестовые программы, выдавая всевозможные сообщения об ошибках

Видимо, ваш компилятор разработан до приема стандарта ANSI и поэтому не способен обрабатывать прототипы функций и тому подобное.

Почему не определены некоторые подпрограммы из стандартной ANSI-библиотеки, хотя у меня ANSI совместимый компилятор?

Нет ничего необычного в том, что компилятор, воспринимающий ANSI синтаксис, не имеет ANSI-совместимых головных файлов или стандартных библиотек.

Почему компилятор «Frobozz Magic C», о котором говорится, что он ANSI-совместимый, не транслирует мою программу? Я знаю, что текст подчиняется стандарту ANSI, потому что он транслируется компилятором gcc

Практически все компиляторы (а **gcc** — более других) поддерживают некоторые нестандартные расширения. Уверены ли вы, что отвергнутый текст не применяет одно из таких расширений? Опасно экспериментировать с компилятором для

исследования языка. Стандарт может допускать отклонения, а компилятор — работать неверно.

Почему мне не удаются арифметические операции с указателем типа void *?

Потому что компилятору не известен размер объекта, на который указывает **void ***. Перед арифметическими операциями используйте оператор приведения к типу (**char ***) или к тому типу, с которым собираетесь работать.

Правильна ли запись a[3]="abc"? Что это значит?

Эта запись верна в ANSI C (и, возможно, в некоторых более ранних компиляторах), хотя полезность такой записи сомнительна. Объявляется массив размера три, инициализируемый тремя буквами **'a'**, **'b'** и **'c'** без завершающего стринг символа **'\0'**. Массив, следовательно, не может использоваться как стринг функциями **strcpy**, **printf %s** и т.п.

Что такое #pragma и где это может пригодиться?

Директива **#pragma** обеспечивает особую, точно определенную «лазейку» для выполнения зависящих от реализации действий: контроль за листингом, упаковку структур, подавление предупреждающих сообщений (вроде комментариев **/* NOTREACHED */** старой программы **lint**) и т.п.

Что означает «#pragma once»? Я нашел эту директиву в одном из головных файлов

Это расширение, реализованное в некоторых препроцессорах, делает головной файл идемпотентным, т.е. эффект от однократного включения файла равен эффекту от многократного включения. Эта директива приводит к тому же результату, что и прием с использованием **#ifndef**.

Вроде бы существует различие между зависимым от реализации, неописанным(unspecified) и неопределенным (undefined) поведением. В чем эта разница?

Если говорить кратко, то при зависимом от реализации поведении необходимо выбрать один вариант и документировать его. При неописанном поведении также выбирается один из вариантов, но в этом случае нет необходимости это документировать. Неопределенное поведение означает, что может произойти все что угодно. Ни в одном из этих случаев Стандарт не выдвигает требований; в первых двух случаях Стандарт иногда

предлагает (а может и требовать) выбор из нескольких близких вариантов поведения.

Если вы заинтересованы в написании мобильных программ, можете игнорировать различия между этими тремя случаями, поскольку всех их необходимо будет избегать.

Как написать макрос для обмена любых двух значений?

На этот вопрос нет хорошего ответа. При обмене целых значений может быть использован хорошо известный трюк с использованием исключающего ИЛИ, но это не сработает для чисел с плавающей точкой или указателей.

Не годится этот прием и в случае, когда оба числа — на самом деле одно и то же число. Из-за многих побочных эффектов не годится и «очевидное» суперкомпактное решение для целых чисел **a^=b^=a^=b**. Когда макрос предназначен для переменных произвольного типа (обычно так и бывает), нельзя использовать временную переменную, поскольку не известен ее тип, а стандартный Си не имеет оператора **typeof**.

Если вы не хотите передавать тип переменной третьим параметров, то, возможно, наиболее гибким, универсальным решением будет отказ от использования макроса.

У меня есть старая программа, которая пытается конструировать идентификаторы с помощью макроса #define Paste(a, b) a//b, но у меня это не работает**

То, что комментарий полностью исчезает, и, следовательно, может быть использован для склеивания соседних лексем (в частности, для создания новых идентификаторов), было недокументированной особенностью некоторых ранних реализаций препроцессора, среди которых заметна была реализация Рейзера (Reiser). Стандарт ANSI, как и K&R, утверждает, что комментарии заменяются единичными пробелами. Но поскольку необходимость склеивания лексем стала очевидной, стандарт ANSI ввел для этого специальный оператор ##, который может быть использован так:

```
#define Paste(a, b) a##b
```

Как наилучшим образом написать cpp макрос, в котором есть несколько инструкций?

Обычно цель состоит в том, чтобы написать макрос, который не отличался бы по виду от функции. Это значит, что

завершающая точка с запятой ставится тем, кто вызывает макрос, а в самом теле макроса ее нет.

Тело макроса не может быть просто составной инструкцией, заключенной в фигурные скобки, поскольку возникнут сообщения об ошибке (очевидно, из-за лишней точки с запятой, стоящей после инструкции) в том случае, когда макрос вызывается после **if**, а в инструкции **if/else** имеется **else**-часть.

Обычно эта проблема решается с помощью такого определения:

```
#define Func() do { \
/* объявления */ \
что-то1; \
что-то2; \
/* ... */ \
} while(0)   /* (нет завершающей ; ) */
```

Когда при вызове макроса добавляется точка с запятой, это расширение становится простой инструкцией вне зависимости от контекста. (Оптимизирующий компилятор удалит излишние проверки или переходы по условию **0**, хотя **lint** это может и не принять.)

Если требуется макрос, в котором нет деклараций или ветвлений, а все инструкции — простые выражения, то возможен другой подход, когда пишется одно, заключенное в круглые скобки выражение, использующее одну или несколько запятых. Такой подход позволяет также реализовать «возврат» значения).

Можно ли в головной файл с помощью #include включить другой головной файл?

Это вопрос стиля, и здесь возникают большие споры. Многие полагают, что «вложенных с помощью **#include** файлов» следует избегать: авторитетный Indian Hill Style Guide неодобрительно отзывается о таком стиле; становится труднее найти соответствующее определение; вложенные **#include** могут привести к сообщениям о многократном объявлении, если головной файл включен дважды; также затрудняется корректировка управляющего файла для утилиты **Make**. С другой стороны, становится возможным использовать модульный принцип при создании головных файлов (головной файл включает с помощью **#include** то, что необходимо только ему; в противном случае придется каждый раз использовать

дополнительный **#include**, что способно вызвать постоянную головную боль); с помощью утилит, подобных **grep** (или файла **tags**) можно легко найти нужные определения вне зависимости от того, где они находятся, наконец, популярный прием:

```
#ifndef HEADERUSED
#define HEADERUSED
...содержимое головного файла...
#endif
```

делает головной файл «идемпотентным», то есть такой файл можно безболезненно включать несколько раз; средства автоматической поддержки файлов для утилиты **Make** (без которых все равно не обойтись в случае больших проектов) легко обнаруживают зависимости при наличии вложенных **#include**.

Работает ли оператор sizeof при использовании средства препроцессора #if?

Нет. Препроцессор работает на ранней стадии компиляции, до того как становятся известны типы переменных. Попробуйте использовать константы, определенные в файле <**limits.h**>, предусмотренном ANSI, или «сконфигурировать» вместо этого командный файл. (А еще лучше написать программу, которая по самой своей природе нечувствительна к размерам переменных).

Можно ли с помощью #if узнать, как организована память машины — по принципу: младший байт — меньший адрес или наоборот?

Видимо, этого сделать нельзя. (Препроцессор использует для внутренних нужд только длинные целые и не имеет понятия об адресации).

А уверены ли вы, что нужно точно знать тип организации памяти? Уж лучше написать программу, которая от этого не зависит.

Во время компиляции мне необходимо сложное препроцесссирование, и я никак не могу придумать, как это сделать с помощью cpp

cpp не задуман как универсальный препроцессор. Чем заставлять **cpp** делать что-то ему не свойственное, подумайте о написании небольшого специализированного препроцессора. Легко раздобыть утилиту типа **make(1)**, которая автоматизирует этот процесс.

Если вы пытаетесь препроцессировать что-то отличное от Си, воспользуйтесь универсальным препроцессором, (таким как **m4**).

Мне попалась программа, в которой, на мой взгляд, слишком много директив препроцессора #ifdef. Как обработать текст, чтобы оставить только один вариант условной компиляции, без использования cpp, а также без раскрытия всех директив #include и #define?

Свободно распространяются программы **unifdef**, **rmifdef** и **scpp**, которые делают в точности то, что вам нужно.

Как получить список предопределенных идентификаторов?

Стандартного способа не существует, хотя необходимость возникает часто. Если руководство по компилятору не содержит этих сведений, то, возможно, самый разумный путь — выделить текстовые строки из исполнимых файлов компилятора или препроцессора с помощью утилиты типа **strings(1)** системы Unix. Имейте в виду, что многие зависящие от системы предопределенные идентификаторы (например, «**unix**») нестандартны (поскольку конфликтуют с именами пользователя) и поэтому такие идентификаторы удаляются или меняются.

Как написать cpp макрос с переменным количеством аргументов?

Популярна такая уловка: определить макрос с одним аргументом, и вызывать его с двумя открывающими и двумя закрывающими круглыми скобками:

```
#define DEBUG(args) (printf("DEBUG: "), printf args)
 if(n != 0) DEBUG(("n is %d\n", n));
```

Очевидный недостаток такого подхода в том, что нужно помнить о дополнительных круглых скобках. Другие решения — использовать различные макросы (DEBUG1, DEBUG2, и т.п.) в зависимости от количества аргументов, или манипулировать запятыми:

```
#define DEBUG(args) (printf("DEBUG: "), printf(args))
#define _ ,
DEBUG("i = %d" _ i)
```

Часто предпочтительнее использовать настоящую функцию, которая стандартным способом может использовать переменное число аргументов.

Как реализовать функцию с переменным числом аргументов?

Используйте головной файл <**stdarg.h**> (или, если необходимо, более старый <**varargs.h**>).

Вот пример функции, которая объединяет произвольное количество стрингов, помещая результат в выделенный с помощью **malloc** участок памяти.

```
#include <stdlib.h>     /* для malloc, NULL, size_t */
#include <stdarg.h>     /* для va_ макросов */
#include <string.h>     /* для strcat и т.п. */

    char *vstrcat(char *first, ...)
    {
          size_t len = 0;
          char *retbuf;
          va_list argp;
          char *p;
          if(first == NULL)
                return NULL;
          len = strlen(first);
                va_start(argp, first);
                while((p = va_arg(argp, char *)) != NULL)
                len += strlen(p);
                va_end(argp);
                retbuf = malloc(len + 1);
              /* +1 для \0 */
                if(retbuf == NULL)
                return NULL;    /* ошибка */
               (void)strcpy(retbuf, first);
                va_start(argp, first);
                while((p = va_arg(argp, char *)) != NULL)
               (void)strcat(retbuf, p);
                va_end(argp);
                return retbuf;
          }
```

Вызывается функция примерно так:

```
char *str = vstrcat("Hello, ", "world!", (char *)NULL);
```

Обратите внимание на явное приведение типа в последнем аргументе. (Помните также, что после вызова такой функции нужно освободить память).

Если компилятор разрабатывался до приема стандарта ANSI, перепишите определение функции без прототипа (**"char *vstrcat(first) char *first; {"**) включите <**stdio.h**> вместо <**stdlib.h**>, добавьте **"extern char *malloc();"**, и используйте **int** вместо **size_t**. Возможно, придется удалить приведение (**void**) и использовать **varargs.h** вместо **stdarg**.

Помните, что в прототипах функций со списком аргументов переменной длины не указывается тип аргументов. Это значит, что по умолчанию будет происходить «расширение» типов аргументов.

Это также значит, что тип нулевого указателя должен быть явно указан.

Как написать функцию, которая бы, подобно printf, получала строку формата и переменное число аргументов, а затем для выполнения большей части работы передавала бы все это printf?

Используйте **vprintf**, **vfprintf** или **vsprintf**.

Перед вами подпрограмма «**error**», которая после строки «**error**:» печатает сообщение об ошибке и символ новой строки.

```
#include <stdio.h>
#include <stdarg.h>

     void
     error(char *fmt, ...)
     {
            va_list argp;
            fprintf(stderr, "error: ");
            va_start(argp, fmt);
            vfprintf(stderr, fmt,    argp);
            va_end(argp);
            fprintf(stderr, "\n");
     }
```

Чтобы использовать старый головной файл <**varargs.h**> вместо <**stdarg.h**>, измените заголовок функции:

```
void error(va_alist)
va_dcl
     {
            char *fmt;
```

измените строку с **va_start**:

```
va_start(argp);
```

и добавьте строку

```
fmt = va_arg(argp, char *);
```

между вызовами **va_start** и **vfprintf**. Заметьте, что после **va_dcl** нет точки с запятой.

Как определить, сколько аргументов передано функции?

Для переносимых программ такая информация недоступна. Некоторые старые системы имели нестандартную функцию **nargs()**, но ее полезность всегда была сомнительна, поскольку обычно эта функция возвращает количество передаваемых машинных слов, а не число аргументов. (Структуры и числа с плавающей точкой обычно передаются в нескольких словах).

Любая функция с переменным числом аргументов должна быть способна по самим аргументам определить их число. Функции типа **printf** определяют число аргументов по спецификаторам формата (%d и т.п.) в строке формата (вот почему эти функции так скверно ведут себя при несовпадении списка аргументов и строки формата). Другой общепринятый прием — использовать признак конца списка (часто это числа **0**, **-1**, или нулевой указатель, приведенный к нужному типу).

Мне не удается добиться того, чтобы макрос va_arg возвращал аргумент типа указатель-на-функцию

Манипуляции с переписыванием типов, которыми обычно занимается **va_arg**, кончаются неудачей, если тип аргумента слишком сложен — вроде указателя на функцию. Если, однако, использовать **typedef** для определения указателя на функцию, то все будет нормально.

Как написать функцию с переменным числом аргументов, которая передает их другой функции с переменным числом аргументов?

В общем случае задача неразрешима. В качестве второй функции нужно использовать такую, которая принимает указатель типа **va_list**, как это делает **vfprintf** в приведенном выше примере. Если аргументы должны передаваться непосредственно (а не через указатель типа **va_list**), и вторая функция принимает переменное число аргументов (и нет возможности создать альтернативную функцию, принимающую указатель **va_list**), то создание переносимой программы невозможно. Проблема может быть решена, если обратиться к языку ассемблера соответствующей машины.

Как вызвать функцию со списком аргументов, создаваемым в процессе выполнения?

Не существует способа, который бы гарантировал переносимость. Если у вас пытливый ум, раздобудьте редактор

таких списков, в нем есть несколько безрассудных идей, которые можно попробовать...

Переменные какого типа правильнее использовать как булевы? Почему в языке Си нет стандартного типа логических переменных? Что использовать для значений true и false — #define или enum?

В языке Си нет стандартного типа логических переменных, потому что выбор конкретного типа основывается либо на экономии памяти, либо на выигрыше времени. Такие вопросы лучше решать программисту (использование типа **int** для булевой переменной может быть быстрее, тогда как использование типа **char** экономит память).

Выбор между **#define** и **enum** — личное дело каждого, и споры о том, что лучше, не особенно интересны.

Используйте любой из вариантов:

```
#define TRUE 1 #define YES 1
#define FALSE 0 #define NO 0
enum bool {false, true}; enum bool {no, yes};
```

или последовательно в пределах программы или проекта используйте числа 1 и 0. (Возможно, задание булевых переменных через **enum** предпочтительнее, если используемый вами отладчик раскрывает содержимое enum-переменных).

Некоторые предпочитают такие способы задания:

```
#define TRUE (1==1)
#define FALSE   (!TRUE)
```

или задают «вспомогательный» макрос:

```
#define Istrue(e) ((e) != 0)
```

Не видно, что они этим выигрывают.

Разве не опасно задавать значение TRUE как 1, ведь в Си любое не равное нулю значение рассматривается как истинное? А если оператор сравнения или встроенный булев оператор возвратит нечто, отличное от 1?

Истинно (да-да!), что любое ненулевое значение рассматривается в Си как значение «ИСТИНА», но это применимо только «на входе», где ожидается булева переменная. Когда булева переменная генерируется встроенным оператором, гарантируется, что она равна 0 или 1.

Следовательно, сравнение

```
if((a == b) == TRUE)
```

как ни смешно оно выглядит, будет вести себя, как ожидается, если значению **TRUE** соответствует 1. Как правило, явные проверки на **TRUE** и **FALSE** нежелательны, поскольку некоторые библиотечные функции (стоит упомянуть **isupper**, **isalpha** и т.п.), возвращают в случае успеха ненулевое значение, которое не обязательно равно 1. (Кроме того, если вы верите, что «**if((a == b) == TRUE)**» лучше чем «**if(a == b)**», то почему не пойти дальше и не написать:

```
if(((a == b) == TRUE) == TRUE)
```

Хорошее «пальцевое» правило состоит в том, чтобы использовать **TRUE** и **FALSE** (или нечто подобное) только когда булевым переменным или аргументам функции присваиваются значения или когда значение возвращается булевой функцией, но никогда при сравнении.

Макроопределения препроцессора **TRUE** и **FALSE** используются для большей наглядности, а не потому, что конкретные значения могут измениться.

Какова разница между enum и рядом директив препроцессора #define?

В настоящее время разница невелика. Хотя многие, возможно, предпочли бы иное решение, стандарт ANSI утверждает, что произвольному числу элементов перечисления могут быть явно присвоены целочисленные значения. (Запрет на присвоение значений без явного приведения типов, позволил бы при разумном использовании перечислений избежать некоторых ошибок.)

Некоторые преимущества перечислений в том, что конкретные значения задаются автоматически, что отладчик может представлять значения перечислимых переменных в символьном виде, а также в том, что перечислимые переменные подчиняются обычным правилам областей действия. (Компилятор может также выдавать предупреждения, когда перечисления необдуманно смешиваются с целочисленными переменными.

Такое смешение может рассматриваться как проявление плохого стиля, хотя формально это не запрещено). Недостаток перечислений в том, что у программиста мало возможностей управлять размером переменных (и предупреждениями компилятора тоже).

Я слышал, что структуры можно копировать как целое, что они могут быть переданы функциям и возвращены ими, но в K&R I сказано, что этого делать нельзя

В K&R I сказано лишь, что ограничения на операции со структурами будут сняты в следующих версиях компилятора; эти операции уже были возможны в компиляторе Денниса Ритчи, когда издавалась книга K&R I.

Хотя некоторые старые компиляторы не поддерживают копирование структур, все современные компиляторы имеют такую возможность, предусмотренную стандартом ANSI C, так что не должно быть колебаний при копировании и передаче структур функциям.

Каков механизм передачи и возврата структур?

Структура, передаваемая функции как параметр, обычно целиком размещается на стеке, используя необходимое количество машинных слов. (Часто для снижения ненужных затрат программисты предпочитают передавать функции указатель на структуру вместо самой структуры).

Структуры часто возвращаются функциями в ту область памяти, на которую указывает дополнительный поддерживаемый компилятором «скрытый» аргумент. Некоторые старые компиляторы используют для возврата структур фиксированную область памяти, хотя это делает невозможным рекурсивный вызов такой функции, что противоречит стандарту ANSI.

Эта программа работает правильно, но после завершения выдает дамп оперативной памяти. Почему?

```
struct list
  {
    char *item;
    struct list *next;
  }
 /* Здесь функция main  */
 main(argc, argv)
 ...
```

Из-за пропущенной точки с запятой компилятор считает, что **main** возвращает структуру. (Связь структуры с функцией **main** трудно определить, мешает комментарий). Так как для возврата структур компилятор обычно использует в качестве скрытого параметра указатель, код, сгенерированный для **main()** пытается принять три аргумента, хотя передаются (в данном случае стартовым кодом Си) только два.

Почему нельзя сравнивать структуры?

Не существует разумного способа сделать сравнение структур совместимым с низкоуровневой природой языка Си. Побайтовое сравнение может быть неверным из-за случайных бит в неиспользуемых «дырках» (такое заполнение необходимо, чтобы сохранить выравнивание для последующих полей). Почленное сравнение потребовало бы неприемлемого количества повторяющихся машинных инструкций в случае больших структур.

Если необходимо сравнить две структуры, напишите для этого свою собственную функцию. C++ позволит создать оператор ==, чтобы связать его с вашей функцией.

Как читать/писать структуры из файла/в файл?

Писать структуры в файл можно непосредственно с помощью **fwrite**:

```
fwrite((char *)&somestruct, sizeof(somestruct), 1, fp);
```

а соответствующий вызов **fread** прочитает структуру из файла.

Однако файлы, записанные таким образом будут не особенно переносимы. Заметьте также, что на многих системах нужно использовать в функции **fopen** флаг «**b**».

Мне попалась программа, в которой структура определяется так:

```
struct name
  {
  int namelen;
  char name[1];
  };
```

затем идут хитрые манипуляции с памятью, чтобы массив name вел себя будто в нем несколько элементов. Такие манипуляции законны/мобильны?

Такой прием популярен, хотя Деннис Ритчи назвал это «слишком фамильярным обращением с реализацией Си». ANSI полагает, что выход за пределы объявленного размера члена структуры не полностью соответствует стандарту, хотя детальное обсуждение всех связанных с этим проблем не входит в задачу данных вопросов и ответов. Похоже, однако, что описанный прием будет одинаково хорошо принят всеми известными реализациями Си. (Компиляторы, тщательно проверяющие границы массивов, могут выдать предупреждения). Для страховки будет лучше объявить переменную очень большого размера чем очень малого. В нашем случае

```
...
  char name[MAXSIZE];
...
```

где **MAXSIZE** больше, чем длина любого имени, которое будет сохранено в массиве **name[]**. (Есть мнение, что такая модификация будет соответствовать Стандарту).

Как определить смещение члена структуры в байтах?

Если это возможно, необходимо использовать макрос **offsetof**, который определен стандартом ANSI. Если макрос отсутствует, предлагается такая (не на все 100% мобильная) его реализация:

```
#define offsetof(type, mem) ((size_t) \
((char *)&((type *) 0)->mem - (char *)((type *) 0)))
```

Для некоторых компиляторов использование этого макроса может оказаться незаконным.

Как осуществить доступ к членам структур по их именам во время выполнения программы?

Создайте таблицу имен и смещений, используя макрос **offsetof()**.

Смещение члена структуры **b** в структуре типа **a** равно:

```
offsetb = offsetof(struct a, b)
```

Если **structp** указывает на начало структуры, а **b** — член структуры типа **int**, смещение которого получено выше, **b** может быть установлен косвенно с помощью:

```
*(int *)((char *)structp + offsetb) = value;
```

Почему sizeof выдает больший размер структурного типа, чем я ожидал, как будто в конце структуры лишние символы?

Это происходит (возможны также внутренние «дыры», когда необходимо выравнивание при задании массива непрерывных структур.

Мой компилятор оставляет дыры в структурах, что приводит к потере памяти и препятствует «двоичному» вводу/выводу при работе с внешними файлами. Могу я отключить «дырообразование» или как-то контролировать выравнивание?

В вашем компиляторе, возможно, есть расширение, (например, **#pragma**), которое позволит это сделать, но стандартного способа не существует.

Можно ли задавать начальные значения объединений?

Стандарт ANSI допускает инициализацию первого члена объединения.

Не существует стандартного способа инициализации других членов (и тем более нет такого способа для старых компиляторов, которые вообще не поддерживают какой-либо инициализации).

Как передать функции структуру, у которой все члены — константы?

Поскольку в языке Си нет возможности создавать безымянные значения структурного типа, необходимо создать временную структуру.

Какой тип целочисленной переменной использовать?

Если могут потребоваться большие числа, (больше 32767 или меньше -32767), используйте тип **long**. Если нет, и важна экономия памяти (большие массивы или много структур), используйте **short**. Во всех остальных случаях используйте **int**. Если важно точно определить момент переполнения и/или знак числа не имеет значения, используйте соответствующий тип **unsigned**. (Но будьте внимательны при совместном использовании типов **signed** и **unsigned** в выражениях). Похожие соображения применимы при выборе между **float** и **double**.

Хотя тип **char** или **unsigned char** может использоваться как целочисленный тип наименьшего размера, от этого больше вреда, чем пользы из-за непредсказуемых перемен знака и возрастающего размера программы.

Эти правила, очевидно, не применимы к адресам переменных, поскольку адрес должен иметь совершенно определенный тип.

Если необходимо объявить переменную определенного размера, (единственной причиной тут может быть попытка удовлетворить внешним требованиям к организации памяти), непременно изолируйте объявление соответствующим **typedef**.

Каким должен быть новый 64-битный тип на новых 64-битных машинах?

Некоторые поставщики Си компиляторов для 64-битных машин поддерживают тип **long int** длиной 64 бита. Другие же, опасаясь, что слишком многие уже написанные программы зависят от **sizeof(int) == sizeof(long) == 32 бита**, вводят новый 64-битный тип **long long** (или **__longlong**).

Программисты, желающие писать мобильные программы, должны, следовательно, изолировать 64-битные типы с помощью средства **typedef**.

Разработчики компиляторов, чувствующие необходимость ввести новый целочисленный тип большего размера, должны объявить его как «имеющий по крайней мере 64 бит» (это действительно новый тип, которого нет в традиционном Си), а не как «имеющий точно 64 бит».

У меня совсем не получается определение связанного списка. Я пишу:

```
typedef struct
{
char *item;
NODEPTR next;
} *NODEPTR;
```

но компилятор выдает сообщение об ошибке. Может структура в Си содержать ссылку на себя?

Структуры в Си, конечно же, могут содержать указатели на себя. В приведенном тексте проблема состоит в том, что определение **NODEPTR** не закончено в том месте, где объявляется член структуры «**next**». Для исправления, снабдите сначала структуру тегом («**struct node**»). Далее объявите «**next**» как «**struct node *next;**», и/или поместите декларацию **typedef** целиком до или целиком после объявления структуры. Одно из возможных решений будет таким:

```
struct node
  {
  char *item;
  struct node *next;
  };
  typedef struct node *NODEPTR;
```

Есть по крайней мере три других одинаково правильных способа сделать то же самое.

Сходная проблема, которая решается примерно так же, может возникнуть при попытке определить с помощью средства **typedef** пару ссылающихся друг на друга структур.

Как объявить массив из N указателей на функции, возвращающие указатели на функции возвращающие указатели на char?

Есть по крайней мере три варианта ответа:

1.

```
char *(*(*a[N])())();
```

2. Писать декларации по шагам, используя **typedef**:

```
typedef char *pc;    /* указатель на char */
typedef pc fpc();    /* функция,возвращающая указатель на
char */
typedef fpc *pfpc;  /* указатель на.. см. выше */
typedef pfpc fpfpc();    /* функция, возвращающая... */
typedef fpfpc *pfpfpc;  /* указатель на... */
pfpfpc a[N];      /* массив... */
```

3. Использовать программу **cdecl,** которая переводит с английского на Си и наоборот.

```
cdecl> declare a as array of pointer to function returning
pointer to function returning pointer to char
char *(*(*a[])())()
```

cdecl может также объяснить сложные декларации, помочь при явном приведении типов, и, для случая сложных деклараций, вроде только что разобранного, показать набор круглых скобок, в которые заключены аргументы. Версии **cdecl** можно найти в comp.sources.unix и в K&R II.

Я моделирую Марковский процесс с конечным числом состояний, и у меня есть набор функций для каждого состояния. Я хочу, чтобы смена состояний происходила путем возврата функцией указателя на функцию, соответствующую следующему состоянию. Однако, я обнаружил ограничение в механизме деклараций языка Си: нет возможности объявить функцию, возвращающую указатель на функцию, возвращающую указатель на функцию, возвращающую указатель на функцию…

Да, непосредственно это сделать нельзя. Пусть функция возвращает обобщенный указатель на функцию, к которому перед вызовом функции будет применен оператор приведения типа, или пусть она возвращает структуру, содержащую только указатель на функцию, возвращающую эту структуру.

Мой компилятор выдает сообщение о неверной повторной декларации, хотя я только раз определил функцию и только раз вызвал

Подразумевается, что функции, вызываемые без декларации в области видимости (или до такой декларации), возвращают значение типа **int**.

Это приведет к противоречию, если впоследствии функция декларирована иначе. Если функция возвращает нецелое значение, она должна быть объявлена до того как будет вызвана.

Как наилучшим образом декларировать и определить глобальные переменные?

Прежде всего заметим, что хотя может быть много деклараций (и во многих файлах) одной «глобальной» (строго говоря «внешней») переменной, (или функции), должно быть всего одно определение. (Определение — это такая декларация, при которой действительно выделяется память для переменной, и присваивается, если нужно, начальное значение). Лучше всего поместить определение в какой-то главный (для программы или ее части). с файл, с внешней декларацией в головном файле **.h**, который при необходимости подключается с помощью **#include**. Файл, в котором находится определение переменной, также должен включать головной файл с внешней декларацией, чтобы компилятор мог проверить соответствие декларации и определения.

Это правило обеспечивает высокую мобильность программ и находится в согласии с требованиями стандарта ANSI C. Заметьте, что многие компиляторы и компоновщики в системе UNIX используют «общую модель», которая разрешает многократные определения без инициализации.

Некоторые весьма странные компиляторы могут требовать явной инициализации, чтобы отличить определение от внешней декларации.

С помощью препроцессорного трюка можно устроить так, что декларация будет сделана лишь однажды, в головном файле, и она с помощью **#define** «превратится» в определение точно при одном включении головного файла.

Что означает ключевое слово extern при декларации функции?

Слово **extern** при декларации функции может быть использовано из соображений хорошего стиля для указания на то,

что определение функции, возможно, находится в другом файле. Формально между:

```
extern int f();
```

и

```
int f();
```

нет никакой разницы.

Я, наконец, понял, как объявлять указатели на функции, но как их инициализировать?

Используйте нечто такое:

```
extern int func();
int (*fp)() = func;
```

Когда имя функции появляется в выражении, но функция не вызывается (то есть, за именем функции не следует "("), оно «сворачивается», как и в случае массивов, в указатель (т.е. неявным образом записанный адрес).

Явное объявление функции обычно необходимо, так как неявного объявления внешней функции в данном случае не происходит (опять-таки из-за того, что за именем функции не следует "(").

Я видел, что функции вызываются с помощью указателей и просто как функции. В чем дело?

По первоначальному замыслу создателя Си указатель на функцию должен был «превратиться» в настоящую функцию с помощью оператора ***** и дополнительной пары круглых скобок для правильной интерпретации.

```
int r, func(), (*fp)() = func;
r = (*fp)();
```

На это можно возразить, что функции всегда вызываются с помощью указателей, но что «настоящие» функции неявно превращаются в указатели (в выражениях, как это происходит при инициализациях) и это не приводит к каким-то проблемам. Этот довод, широко распространенный компилятором pcc и принятый стандартом ANSI, означает, что выражение:

```
r = fp();
```

работает одинаково правильно, независимо от того, что такое **fp** — функция или указатель на нее. (Имя всегда используется однозначно; просто невозможно сделать что-то другое с

указателем на функцию, за которым следует список аргументов, кроме как вызвать функцию.)

Явное задание * безопасно и все еще разрешено (и рекомендуется, если важна совместимость со старыми компиляторами).

Где может пригодиться ключевое слово auto?

Нигде, оно вышло из употребления.

Что плохого в таких строках:
char c;
while((c = getchar())!= EOF)...

Во-первых, переменная, которой присваивается возвращенное **getchar** значение, должна иметь тип **int. getchar** и может вернуть все возможные значения для символов, в том числе EOF. Если значение, возвращенное **getchar** присваивается переменной типа **char**, возможно либо обычную литеру принять за EOF, либо EOF исказится (особенно если использовать тип **unsigned char**) так, что распознать его будет невозможно.

Как напечатать символ '%' в строке формата printf? Я попробовал \%, но из этого ничего не вышло

Просто удвойте знак процента %%.

Почему не работает scanf("%d",i)?

Для функции **scanf** необходимы адреса переменных, по которым будут записаны данные, нужно написать **scanf("%d", &i);**

Почему не работает
double d;
scanf("%f", &d);

scanf использует спецификацию формата **%lf** для значений типа **double** и **%f** для значений типа **float**. (Обратите внимание на несходство с **printf**, где в соответствии с правилом расширения типов аргументов спецификация **%f** используется как для **float**, так и для **double**).

Почему фрагмент программы
while(!feof(infp)) {
fgets(buf, MAXLINE, infp);
fputs(buf, outfp);
}
дважды копирует последнюю строку?

Это вам не Паскаль. Символ EOF появляется только после попытки прочесть, когда функция ввода натыкается на конец файла.

Чаще всего необходимо просто проверять значение, возвращаемое функцией ввода, (в нашем случае **fgets**); в использовании **feof()** обычно вообще нет необходимости.

Почему все против использования gets()?

Потому что нет возможности предотвратить переполнение буфера, куда читаются данные, ведь функции **gets()** нельзя сообщить его размер.

Почему переменной errno присваивается значение ENOTTY после вызова printf()?

Многие реализации стандартной библиотеки ввода/вывода несколько изменяют свое поведение, если стандартное устройство вывода — терминал. Чтобы определить тип устройства, выполняется операция, которая оканчивается неудачно (с сообщением **ENOTTY**), если устройство вывода — не терминал. Хотя вывод завершается успешно, **errno** все же содержит **ENOTTY**.

Запросы моей программы, а также промежуточные результаты не всегда отображаются на экране, особенно когда моя программа передает данные по каналу (pipe) другой программе

Лучше всего явно использовать **fflush(stdout)**, когда непременно нужно видеть то, что выдает программа. Несколько механизмов пытаются «в нужное время» осуществить **fflush**, но, похоже, все это правильно работает в том случае, когда **stdout** — это терминал.

При чтении с клавиатуры функцией scanf возникает чувство, что программа зависает, пока я перевожу строку

Функция **scanf** была задумана для ввода в свободном формате, необходимость в котором возникает редко при чтении с клавиатуры.

Что же касается ответа на вопрос, то символ «**\n**» в форматной строке вовсе не означает, что **scanf** будет ждать

перевода строки. Это значит, что **scanf** будет читать и отбрасывать все встретившиеся подряд пробельные литеры (т.е. символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и новой страницы).

Похожее затруднение случается, когда **scanf** «застревает», получив неожиданно для себя нечисловые данные. Из-за подобных проблем часто лучше читать всю строку с помощью **fgets**, а затем использовать **sscanf** или другие функции, работающие со строками, чтобы интерпретировать введенную строку по частям. Если используется **sscanf**, не забудьте проверить возвращаемое значение для уверенности в том, что число прочитанных переменных равно ожидаемому.

Я пытаюсь обновить содержимое файла, для чего использую fopen в режиме «r+», далее читаю строку, затем пишу модифицированную строку в файл, но у меня ничего не получается

Непременно вызовите **fseek** перед записью в файл. Это делается для возврата к началу строки, которую вы хотите переписать; кроме того, всегда необходимо вызвать **fseek** или **fflush** между чтением и записью при чтении/записи в режимах «+». Помните также, что литеры можно заменить лишь точно таким же числом литер.

Как мне отменить ожидаемый ввод, так, чтобы данные, введенные пользователем, не читались при следующем запросе? Поможет ли здесь fflush(stdin)?

fflush определена только для вывода. Поскольку определение «**flush**» («смывать») означает завершение записи символов из буфера (а не отбрасывание их), непрочитанные при вводе символы не будут уничтожены с помощью **fflush**. Не существует стандартного способа игнорировать символы, еще не прочитанные из входного буфера **stdio**. Не видно также, как это вообще можно сделать, поскольку непрочитанные символы могут накапливаться в других, зависящих от операционной системы, буферах.

Как перенаправить stdin или stdout в файл?

Используйте **freopen**.

Если я использовал freopen, то как вернуться назад к stdout (stdin)?

Если необходимо переключаться между **stdin (stdout)** и файлом, наилучшее универсальное решение — не спешить использовать **freopen**.

Попробуйте использовать указатель на файл, которому можно по желанию присвоить то или иное значение, оставляя значение **stdout (stdin)** нетронутым.

Как восстановить имя файла по указателю на открытый файл?

Это проблема, вообще говоря, неразрешима. В случае операционной системы UNIX, например, потребуется поиск по всему диску (который, возможно, потребует специального разрешения), и этот поиск окончится неудачно, если указатель на файл был каналом (pipe) или был связан с удаленным файлом. Кроме того, обманчивый ответ будет получен для файла со множественными связями. Лучше всего самому запоминать имена при открытии файлов (возможно, используя специальные функции, вызываемые до и после **fopen**).

Почему strncpy не всегда завершает строку-результат символом '\0'?

strncpy была задумана для обработки теперь уже устаревших структур данных — «строк» фиксированной длины, не обязательно завершающихся символом '\0'. И, надо сказать, **strncpy** не совсем удобно использовать в других случаях, поскольку часто придется добавлять символ '\0' вручную.

Я пытаюсь сортировать массив строк с помощью qsort, используя для сравнения strcmp, но у меня ничего не получается

Когда вы говорите о «массиве строк», то, видимо, имеете в виду «массив указателей на **char**». Аргументы функции сравнения, работающей в паре с **qsort** — это указатели на сравниваемые объекты, в данном случае — указатели на указатели на **char**. (Конечно, **strcmp** работает просто с указателями на **char**).

Аргументы процедуры сравнения описаны как «обобщенные указатели» **const void *** или **char ***. Они должны быть превращены в то, что они представляют на самом деле, т.е. (**char ****) и дальше нужно раскрыть ссылку с помощью *****; тогда **strcmp** получит именно то, что нужно для сравнения. Напишите функцию сравнения примерно так:

```
int pstrcmp(p1, p2)  /* сравнить строки, используя указатели
*/
char *p1, *p2; /* const void * для ANSI C */
  {
     return strcmp(*(char **)p1, *(char **)p2);
  }
```

Сейчас я пытаюсь сортировать массив структур с помощью qsort. Процедура сравнения, которую я использую, принимает в качестве аргументов указатели на структуры, но компилятор выдает сообщение о неверном типе функции сравнения. Как мне преобразовать аргументы функции, чтобы подавить сообщения об ошибке?

Преобразования должны быть сделаны внутри функции сравнения, которая должна быть объявлена как принимающая аргументы типа «обобщенных указателей (**const void *** или **char ***).

Функция сравнения может выглядеть так:

```
int mystructcmp(p1, p2)
char *p1, *p2;    /* const void * для ANSI C */
    {
  struct mystruct *sp1 = (struct mystruct *)p1;
  struct mystruct *sp2 = (struct mystruct *)p2;
  /* теперь сравнивайте sp1->что-угодно и sp2-> ... */
    }
```

С другой стороны, если сортируются указатели на структуры, необходима косвенная адресация:

```
sp1 = *(struct mystruct **)p1
```

Как преобразовать числа в строки (операция, противоположная atoi)? Есть ли функция itoa?

Просто используйте **sprintf**. (Необходимо будет выделить память для результата. Беспокоиться, что **sprintf** — слишком сильное средство, которое может привести к перерасходу памяти и увеличению времени выполнения, нет оснований. На практике **sprintf** работает хорошо).

Как получить дату или время в Си программе?

Просто используйте функции **time**, **ctime**, и/или **localtime**. (Эти функции существуют многие годы,они включены в стандарт ANSI).

Вот простой пример:

```
#include <stdio.h>
#include <time.h>
main()
{
  time_t now = time((time_t *)NULL);
  printf("It's %.24s.\n", ctime(&now));
  return 0;
  }
```

Я знаю, что библиотечная функция localtime разбивает значение time_t по отдельным членам структуры tm, а функция ctime превращает time_t в строку символов. А как проделать обратную операцию перевода структуры tm или строки символов в значение time_t?

Стандарт ANSI определяет библиотечную функцию **mktime**, которая преобразует структуру **tm** в **time_t**. Если ваш компилятор не поддерживает **mktime**, воспользуйтесь одной из общедоступных версий этой функции.

Перевод строки в значение **time_t** выполнить сложнее из-за большого количества форматов дат и времени, которые должны быть распознаны.

Некоторые компиляторы поддерживают функцию **strptime**; другая популярная функция — **partime** широко распространяется с пакетом RCS, но нет уверенности, что эти функции войдут в Стандарт.

Как прибавить n дней к дате? Как вычислить разность двух дат?

Вошедшие в стандарт ANSI/ISO функции **mktime** и **difftime** могут помочь при решении обеих проблем. **mktime()** поддерживает ненормализованные даты, т.е. можно прямо взять заполненную структуру **tm**, увеличить или уменьшить член **tm_mday**, затем вызвать **mktime()**, чтобы нормализовать члены **year**, **month** и **day** (и преобразовать в значение **time_t**).

difftime() вычисляет разность в секундах между двумя величинами типа **time_t**. **mktime()** можно использовать для вычисления значения **time_t** разности двух дат. (Заметьте, однако, что все эти приемы возможны лишь для дат, которые могут быть представлены значением типа **time_t**; кроме того, из-за переходов на летнее и зимнее время продолжительность дня не точно равна 86400 сек.).

Мне нужен генератор случайных чисел

В стандартной библиотеке Си есть функция rand(). Реализация этой функции в вашем компиляторе может не быть идеальной, но и создание лучшей функции может оказаться очень непростым.

Как получить случайные целые числа в определенном диапазоне?

Очевидный способ:

```
rand() % N
```

где **N**, конечно, интервал, довольно плох, ведь поведение

младших бит во многих генераторах случайных чисел огорчает своей неслучайностью. Лучше попробуйте нечто вроде:

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
```

Если вам не нравится употребление чисел с плавающей точкой, попробуйте:

```
rand() / (RAND_MAX / N + 1)
```

Оба метода требуют знания **RAND_MAX** (согласно ANSI, **RAND_MAX** определен в <**stdlib.h**>. Предполагается, что **N** много меньше **RAND_MAX**.

Каждый раз при запуске программы функция rand() выдает одну и ту же последовательность чисел

Можно вызвать **srand()** для случайной инициализации генератора случайных чисел. В качестве аргумента для **srand()** часто используется текущее время, или время, прошедшее до нажатия на клавишу (хотя едва ли существует мобильная процедура определения времен нажатия на клавиши).

Мне необходима случайная величина, имеющая два значения true/false. Я использую rand() % 2, но получается неслучайная последовательность: 0,1,0,1,0...

Некачественные генераторы случайных чисел (попавшие, к несчастью, в состав некоторых компиляторов) не очень то случайны, когда речь идет о младших битах. Попробуйте использовать старшие биты.

Я все время получаю сообщения об ошибках — не определены библиотечные функции, но я включаю все необходимые головные файлы

Иногда (особенно для нестандартных функций) следует явно указывать, какие библиотеки нужны при компоновке программы.

Я по-прежнему получаю сообщения, что библиотечные функции не определены, хотя и использую ключ -l, чтобы явно указать библиотеки во время компоновки

Многие компоновщики делают один проход по списку объектных файлов и библиотек, которые вы указали, извлекая из библиотек только те функции, удовлетворяющие ссылки, которые к этому моменту оказались неопределенными. Следовательно, порядок относительно объектных файлов, в котором перечислены библиотеки, важен; обычно просмотр библиотек нужно делать в самом конце. (Например, в операционной системе UNIX помещайте ключи -l в самом конце командной строки).

Мне необходим исходный текст программы, которая осуществляет поиск заданной строки

Ищите библиотеку **regesp** (поставляется со многими UNIX-системами) или достаньте пакет **regexp** Генри Спенсера (Henry Spencer).

Как разбить командную строку на разделенные пробельными литерами аргументы (что-то вроде argc и argv в main)?

В большинстве компиляторов имеется функция **strtok**, хотя она требует хитроумного обращения, а ее возможности могут вас не удовлетворить (например, работа в случае кавычек).

Вот я написал программу, а она ведет себя странно. Что в ней не так?

Попробуйте сначала запустить **lint** (возможно, с ключами **-a**, **-c**, **-h**, **-p**). Многие компиляторы Си выполняют на самом деле только половину задачи, не сообщая о тех подозрительных местах в тексте программы, которые не препятствуют генерации кода.

Как мне подавить сообщение «warning: possible pointer alignment problem» («предупреждение: возможна проблема с выравниванием указателя»), которое выдает lint после каждого вызова malloc?

Проблема состоит в том, что **lint** обычно не знает, и нет возможности ему об этом сообщить, что **malloc** «возвращает указатель на область памяти, которая должным образом выровнена для хранения объекта любого типа». Возможна псевдореализация **malloc** с помощью **#define** внутри **#ifdef lint**, которая удалит это сообщение, но слишком прямолинейное применение **#define** может подавить и другие осмысленные сообщения о действительно некорректных вызовах. Возможно, будет проще игнорировать эти сообщения, может быть, делать это автоматически с помощью **grep -v**.

Где найти ANSI-совместимый lint?

Программа, которая называется **FlexeLint** (в виде исходного текста с удаленными комментариями и переименованными переменными, пригодная для компиляции на «почти любой» системе) может быть заказана по адресу:

```
Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426 USA
(+1) 610 584 4261
gimpel@netaxs.com
```

Lint для System V release 4 ANSI-совместим и может быть получен (вместе с другими С утилитами) от UNIX Support Labs или от дилеров System V.

Другой ANSI-совместимый **LINT** (способный также выполнять формальную верификацию высокого уровня) называется **LCLint** и доступен через:
ftp: larch.lcs.mit.edu://pub/Larch/lclint/.

Ничего страшного, если программы **lint** нет. Многие современные компиляторы почти столь же эффективны в выявлении ошибок и подозрительных мест, как и **lint**.

Может ли простой и приятный трюк if(!strcmp(s1, s2)) служить образцом хорошего стиля?

Стиль не особенно хороший, хотя такая конструкция весьма популярна.

Тест удачен в случае равенства строк, хотя по виду условия можно подумать, что это тест на неравенство.

Есть альтернативный прием, связанный с использованием макроса:

```
#define Streq(s1, s2) (strcmp((s1), (s2)) == 0)
```

Вопросы стиля программирования, как и проблемы веры, могут обсуждаться бесконечно. К хорошему стилю стоит стремиться, он легко узнаваем, но не определим.

Каков наилучший стиль внешнего оформления программы?

Не так важно, чтобы стиль был «идеален». Важнее, чтобы он применялся последовательно и был совместим (со стилем коллег или общедоступных программ).

Так трудно определимое понятие «хороший стиль» включает в себя гораздо больше, чем просто внешнее оформление программы; не тратьте слишком много времени на отступы и скобки в ущерб более существенным слагаемым качества.

У меня операции с плавающей точкой выполняются странно, и на разных машинах получаются различные результаты

Сначала убедитесь, что подключен головной файл <**math.h**> и правильно объявлены другие функции, возвращающие тип **double**.

Если дело не в этом, вспомните, что большинство компьютеров используют форматы с плавающей точкой, которые хотя и похоже, но вовсе не идеально имитируют операции с действительными числами. Потеря значимости, накопление ошибок и другие свойственные ЭВМ особенности вычислений могут быть весьма болезненными.

Не нужно предполагать, что результаты операций с плавающей точкой будут точными, в особенности не стоит проверять на равенство два числа с плавающей точкой. (Следует избегать любых ненужных случайных факторов.)

Все эти проблемы одинаково свойственны как Си, так и другим языкам программирования. Семантика операций с плавающей точкой определяется обычно так, «как это выполняет процессор»; иначе компилятор вынужден бы был заниматься непомерно дорогостоящей эмуляцией «правильной» модели вычислений.

Я пытаюсь проделать кое-какие вычисления, связанные с тригонометрией, включаю <math.h>, но все равно получаю сообщение: «undefined: _sin» во время компиляции

Убедитесь в том, что компоновщику известна библиотека, в которой собраны математические функции. Например, в операционной системе UNIX часто необходим ключ **-lm** в самом конце командной строки.

Почему в языке Си нет оператора возведения в степень?

Потому что немногие процессоры имеют такую инструкцию. Вместо этого можно, включив головной файл <**math.h**>, использовать функцию **pow()**, хотя часто при небольших целых порядках явное умножение предпочтительней.

Как округлять числа?

Вот самый простой и честный способ:

```
(int)(x + 0.5)
```

Хотя для отрицательных чисел это не годится.

Как выявить специальное значение IEEE NaN и другие специальные значения?

Многие компиляторы с высококачественной реализацией стандарта **IEEE** операций с плавающей точкой обеспечивают возможность (например, макрос **isnan()**) явной работы с такими значениями, а Numerical C Extensions Group (NCEG) занимается

стандартизацией таких средств. Примером грубого, но обычно эффективного способа проверки на **NaN** служит макрос:

```
#define isnan(x) ((x) != (x))
```

хотя не знающие об **IEEE** компиляторы могут выбросить проверку в процессе оптимизации.

У меня проблемы с компилятором Turbo C. Программа аварийно завершается, выдавая нечто вроде «floating point formats not linked»

Некоторые компиляторы для мини-эвм, включая Turbo C (а также компилятор Денниса Ритчи для PDP-11), не включают поддержку операций с плавающей точкой, когда им кажется, что это не понадобится.

В особенности это касается версий **printf** и **scanf**, когда для экономии места не включается поддержка **%e**, **%f** и **%g**. Бывает так, что эвристической процедуры Turbo C, которая определяет — использует программа операции с плавающей точкой или нет, оказывается недостаточно, и программист должен лишний раз вызвать функцию, использующую операции с плавающей точкой, чтобы заставить компилятор включить поддержку таких операций.

Как прочитать с клавиатуры один символ, не дожидаясь новой строки?

Вопреки популярному убеждению и желанию многих, этот вопрос (как и родственные вопросы, связанные с дублированием символов) не относится к языку Си. Передача символов с «клавиатуры» программе, написанной на Си, осуществляется операционной системой, эта операция не стандартизирована языком Си. Некоторые версии библиотеки **curses** содержат функцию **cbreak()**, которая делает как раз то, что нужно.

Если вы пытаетесь прочитать пароль с клавиатуры без вывода его на экран, попробуйте **getpass()**. В операционной системе UNIX используйте **ioctl** для смены режима работы драйвера терминала (**CBREAK** или **RAW** для «классических» версий; **ICANON**, **c_cc[VMIN]** и **c_cc[VTIME]** для System V или Posix). В системе MS-DOS используйте **getch()**. В системе VMS попробуйте функции управления экраном (**SMG$**) или **curses**, или используйте низкоуровневые команды **$QIO** с кодами **IO$_READVBLK** (и, может быть, **IO$M_NOECHO**) для приема одного символа за раз. В других операционных системах выкручивайтесь сами. Помните, что в некоторых операционных системах сделать нечто подобное невозможно, так как работа с

символами осуществляется вспомогательными процессорами и не находится под контролем центрального процессора.

Вопросы, ответы на которые зависят от операционной системы, неуместны в **comp.lang.c**. Ответы на многие вопросы можно найти в FAQ таких групп как **comp.unix.questions** и **comp.os.msdos.programmer**.

Имейте в виду, что ответы могут отличаться даже в случае разных вариантов одной и той же операционной системы. Если вопрос касается специфики операционной системы, помните, что ответ, пригодный в вашей системе, может быть бесполезен всем остальным.

Как определить — есть ли символы для чтения (и если есть, то сколько?) И наоборот, как сделать, чтобы выполнение программы не блокировалось, когда нет символов для чтения?

Ответ на эти вопросы также целиком зависит от операционной системы.

В некоторых версиях **curses** есть функция **nodelay()**. В зависимости от операционной системы вы сможете использовать «неблокирующий ввод/вывод» или системный вызов «**select**» или **ioctl FIONREAD**, или **kbhit()**, или **rdchk()**, или опцию **O_NDELAY** функций **open()** или **fcntl()**.

Как очистить экран? Как выводить на экран негативное изображение?

Это зависит от типа терминала (или дисплея). Можете использовать такую библиотеку как **termcap** или **curses**, или какие-то другие функции, пригодные для данной операционной системы.

Как программа может определить полный путь к месту, из которого она была вызвана?

argv[0] может содержать весь путь, часть его или ничего не содержать.

Если имя файла в **argv[0]** имеется, но информация не полна, возможно повторение логики поиска исполнимого файла, используемой интерпретатором командного языка. Гарантированных или мобильных решений, однако, не существует.

Как процесс может изменить переменную окружения родительского процесса?

В общем, никак. Различные операционные системы обеспечивают сходную с UNIX возможность задания пары имя/значение. Может ли программа с пользой для себя поменять окружение, и если да, то как — все это зависит от операционной системы.

В системе UNIX процесс может модифицировать свое окружение (в некоторых системах есть для этого функции **setenv()** и/или **putenv()**) и модифицированное окружение обычно передается дочерним процессам но не распространяется на родительский процесс.

Как проверить, существует ли файл? Мне необходимо спрашивать пользователя перед тем как переписывать существующие файлы

В UNIX-подобных операционных системах можно попробовать функцию **access()**, хотя имеются кое-какие проблемы. (Применение **access()** может сказаться на последующих действиях, кроме того, возможны особенности исполнения в setuid-программах). Другое (возможно, лучшее) решение — вызвать **stat()**, указав имя файла. Единственный универсальный, гарантирующий мобильность способ состоит в попытке открыть файл.

Как определить размер файла до его чтения?

Если «размер файла» — это количество литер, которое можно прочитать, то, вообще говоря, это количество заранее неизвестно. В операционной системе Unix вызов функции **stat** дает точный ответ, и многие операционные системы поддерживают похожую функцию, которая дает приблизительный ответ. Можно с помощью **fseek** переместиться в конец файла, а затем вызвать **ftell**, но такой прием немобилен (дает точный ответ только в системе Unix, в других же случаях ответ почти точен лишь для определенных стандартом ANSI «двоичных» файлов).

В некоторых системах имеются подпрограммы **filesize** или **filelength**.

И вообще, так ли нужно заранее знать размер файла? Ведь самый точный способ определения его размера в Си программе заключается в открытии и чтении. Может быть, можно изменить программу так, что размер файла будет получен в процессе чтения?

Как укоротить файл без уничтожения или переписывания?

В системах BSD есть функция **ftruncate()**, несколько других систем поддерживают **chsize()**, в некоторых имеется (возможно, недокументированный) параметр **fcntl F_FREESP**. В системе MS-DOS можно иногда использовать **write(fd, "", 0)**. Однако, полностью мобильного решения не существует.

Как реализовать задержку или определить время реакции пользователя, чтобы погрешность была меньше секунды?

У этой задачи нет, к несчастью, мобильных решений. Unix V7 и ее производные имели весьма полезную функцию **ftime()** с точностью до миллисекунды, но она исчезла в System V и Posix. Поищите такие функции: **nap()**, **setitimer()**, **msleep()**, **usleep()**, **clock()** и **gettimeofday()**. Вызовы **select()** и **poll()** (если эти функции доступны) могут быть добавлены к сервисным функциям для создания простых задержек. В системе MS-DOS возможно перепрограммирование системного таймера и прерываний таймера.

Как прочитать объектный файл и передать управление на одну из его функций?

Необходим динамический компоновщик и/или загрузчик. Возможно выделить память с помощью malloc и читать объектные файлы, но нужны обширные познания в форматах объектных файлов, модификации адресов и пр.

В системе BSD Unix можно использовать **system()** и **ld -A** для динамической компоновки. Многие (большинство) версии SunOS и System V имеют библиотеку **-ldl**, позволяющую динамически загружать объектные модули. Есть еще GNU пакет, который называется «**dld**».

Как выполнить из программы команду операционной системы?

Используйте **system()**.

Как перехватить то, что выдает команда операционной системы?

Unix и некоторые другие операционные системы имеют функцию **popen()**, которая переназначает поток **stdio** каналу, связанному с процессом, запустившим команду, что позволяет прочитать выходные данные (или передать входные). А можно просто перенаправить выход команды в файл, затем открыть его и прочесть.

Как получить содержимое директории в Си программе?

Выясните, нельзя ли использовать функции **opendir()** и **readdir()**, доступные в большинстве систем Unix. Реализации этих функций известны для MS-DOS, VMS и других систем. (MS-DOS имеет также функции **findfirst** и **findnext**, которые делают в точности то же самое).

Как работать с последовательными (COM) портами?

Это зависит от операционной системы. В системе Unix обычно осуществляются операции открытия, чтения и записи во внешнее устройство и используются возможности терминального драйвера для настройки характеристик. В системе MS-DOS можно либо использовать прерывания BIOSa, либо (если требуется приличная скорость) один из управляемых прерываниями пакетов для работы с последовательными портами.

Что можно с уверенностью сказать о начальных значениях переменных, которые явным образом не инициализированы? Если глобальные переменные имеют нулевое начальное значение, то правильно ли нулевое значение присваивается указателям и переменным с плавающей точкой?

«Статические» переменные (то есть объявленные вне функций и те, что объявлены как принадлежащие классу **static**) всегда инициализируются (прямо при старте программы) нулем, как будто программист написал «**=0**». Значит, переменные будут инициализированы как нулевые указатели (соответствующего типа), если они объявлены указателями, или значениями **0.0**, если были объявлены переменные с плавающей точкой.

Переменные автоматического класса (т.е. локальные переменные без спецификации **static**), если они явно не определены, первоначально содержат «мусор». Никаких полезных предсказаний относительно мусора сделать нельзя.

Память, динамически выделяемая с помощью **malloc** и **realloc** также будет содержать мусор и должна быть инициализирована, если это необходимо, вызывающей программой. Память, выделенная с помощью **calloc**, зануляет все биты, что не всегда годится для указателей или переменных с плавающей точкой.

Этот текст взят прямо из книги, но он не компилируется:

```
f()
{
char a[] = "Hello, world!";
}
```

Возможно, ваш компилятор создан до принятия стандарта ANSI и еще не поддерживает инициализацию «автоматических агрегатов» (то есть нестатических локальных массивов и структур).

Чтобы выкрутиться из этой ситуации, сделайте массив статическим или глобальным, или инициализируйте его с помощью **strcpy**, когда вызывается **f()**. (Всегда можно инициализировать автоматическую переменную **char *** стрингом литер.)

Как писать данные в файл, чтобы их можно было читать на машинах с другим размером слова, порядком байтов или другим форматом чисел с плавающей точкой?

Лучшее решение — использовать текстовые файлы (обычно ASCII), с данными, записанными **fprintf**. Читать данные лучше всего с помощью **fscanf** или чего-то подобного. (Такой же совет применим для сетевых протоколов). К мнениям, что текстовые файлы слишком велики и могут долго обрабатываться, относитесь скептически.

Помимо того, что эффективность таких операций может быть на практике приемлемой, способность манипулировать данными с помощью стандартных средств может иметь решающее значение.

Если необходимо использовать двоичный формат, переносимость данных можно улучшить (или получить выгоду от использования готовых библиотек ввода/вывода), если использовать стандартные форматы данных, такие как XDR (RFC 1014) (Sun), ASN.1(OSI), X.409 (CCITT), или ISO 8825 «Основные правила кодирования».

Как вставить или удалить строку (или запись) в середине файла?

Придется, видимо, переписать файл.

Как возвратить из функции несколько значений?

Или передайте указатель на то место, которое будет заполнено функцией, или пусть функция возвращает структуру,

содержащую желаемые значения, или подумайте о глобальных переменных (если их немного).

Если есть указатель (char *) на имя функции в виде стринга, то как эту функцию вызвать?

Наиболее прямолинейный путь — создание таблицы имен и соответствующих им указателей:

```
int function1(), function2();
struct {char *name; int (*funcptr)(); } symtab[] =
{
"function1",   function1,
"function2",   function2,
};
```

Ну а теперь нужно поискать в таблице нужное имя и вызвать функцию, используя связанный с именем указатель.

У меня, кажется, нет головного файла <sgtty.h>. Где мне его взять?

Стандартные головные файлы существуют в том смысле, что содержат информацию, необходимую компилятору, операционной системе и процессору. «Чужой» головной файл подойдет лишь тогда, когда взят из идентичного окружения. Поинтересуйтесь у продавца компилятора, почему отсутствует головной файл, или попросите прислать новый взамен потерянного.

Как вызвать процедуры, написанные на языке FORTRAN (C++,BASIC,Pascal, Ada, Lisp) из Си (и наоборот)?

Ответ полностью зависит от машины и от специфики передачи параметров различными компиляторами. Решения вообще может не быть. Внимательно читайте руководство по компилятору. Иногда в документации имеется «Руководство по смешанному программированию», хотя техника передачи аргументов и обеспечения правильного входа в функцию зачастую весьма таинственна. Дополнительная информация находится в файле FORT.gz Глена Гирса, (Glenn Geers) который можно получить с помощью ftp suphys.physics.su.oz.au в директории src.

Головной файл cfortran.h упрощает взаимодействие C/FORTRAN на многих популярных машинах. cfortran.h можно получит через ftp zebra.desy.de (131.169.2.244).

В C++ модификатор «C» внешней функции показывает, что функция будет вызываться с использованием соглашения о передаче параметров языка Си.

Кто-нибудь знает о программах, переводящих Pascal или FORTRAN (или LISP, Ada, awk, «старый» Си) в Си?

Есть несколько общедоступных программ:

- **p2c** — переводчик с Паскаля на Си, написанный Дейвом Гиллеспи, (Dave Gillespie) помещен в comp.sources.unix в Марте 1990 (Volume 21); доступен также через ftp csvax.cs.caltech.edu, файл pub/p2c-1.20.tar.Z.
- **ptoc** — другой переводчик с Паскаля на Си, написан на Паскале (comp.sources.unix, Volume 10, поправки в vol. 13?)
- **f2c** — переводчик с фортрана на Си совместно разработанный Bell Labs, Bellcore, and Carnegie Mellon. Подробности можно получить, послав электронной почтой сообщение «send index from f2c» по адресу netlib@research.att.com или research!netlib. (Эти подробности можно получить и через ftp netlib.att.com, в директории netlib/f2c.)

Составитель этого списка вопросов и ответов имеет список других коммерческих трансляторов, среди них трансляторы для менее известных языков.

Правда ли, что C++ — надмножество Си. Можно ли использовать компилятор C++ для трансляции C программ?

Си++ вырос из Си и в большой степени базируется на нем, но некоторые правильные конструкции Си недопустимы в C++. (Многие Си программы, будут, тем не менее, правильно транслироваться компилятором Си++).

Мне нужен генератор перекрестных ссылок Си и C форматизатор

Ищи программы, которые называются **cflow**, **calls**, **cscope**, **cb**, **indent**.

Где найти все эти общедоступные программы?

Если у вас есть доступ к Usenet, смотрите периодически помещаемые сообщения в comp.sources.unix и comp.sources.misc, которые описывают некоторые детали ведения архивов и подсказывают, как получить те или иные файлы. Обычно используется **ftp** и/или **uucp** с центральным, ориентированным на

пользователей сервером, таким как **uunet** (ftp.uu.net, 192.48.96.9). Однако, в этих вопросах и ответах невозможно исследовать или перечислить все архивные серверы и рассказать о доступе к ним.

Ай Ша (Ajay Shah) поддерживает список общедоступных программ в области численного анализа, который периодически публикуется, и его можно найти там же, где и данные вопросы и ответы. Группа Usenet comp.archives содержит многочисленные объявления о том, что доступно на различных ftp. Почтовый сервер «archie» может подсказать, на каком ftp имеются те или иные программы.

Наконец, группа comp.sources.wanted — обычно самое подходящее место, где можно поместить соответствующий запрос, но посмотрите прежде их список вопросов и ответов (FAQ) «Как найти источники».

Почему недопустимы вложенные комментарии? Как прикажете «выключить» фрагмент программы, в котором уже есть комментарии? Можно ли использовать комментарии внутри стринговых констант?

Вложенные комментарии принесут больше вреда, чем пользы, главным образом из-за возможности случайно не закрыть комментарий, оставив внутри него символы «/*». По этой причине лучше «выключить» большой фрагмент программы, в котором уже есть комментарии, с помощью средств препроцессора **#ifdef** или **#if 0**.

Последовательность символов /* и */ не имеет специального значения внутри заключенных в двойные кавычки стрингов. Эта последовательность не рассматривается как комментарий, поскольку программа (особенно та, которая создает текст другой Си программы) должна иметь возможность эти комментарии печатать.

Как получить значение кода ASCII той или иной литеры, и наоборот?

В Си литеры представлены целыми числами, соответствующими их значениям (в соответствии с набором символов данной машины). Так что нет необходимости в преобразовании: если известна литера, то известно и ее значение.

Как реализовать последовательности и/или массивы бит?

Используйте массивы переменных типа **char** или **int** и несколько макросов для операций с отдельными битами

(используйте определение 8 для **CHAR_BIT**, если нет головного файла **<limits.h>**:

```
#include <limits.h>    /* для CHAR_BIT */
#define BITMASK(bit) (1 << ((bit) % CHAR_BIT))
   #define BITSLOT(bit) ((bit) / CHAR_BIT)
   #define BITSET(ary, bit) ((ary)[BITSLOT(bit)] |=
BITMASK(bit))
  #define BITTEST(ary, bit) ((ary)[BITSLOT(bit)] &
BITMASK(bit))
```

Как наилучшим образом определить число установленных бит, соответствующих определенному значению?

Решение этой и многих других проблем из области битоверчения можно ускорить и сделать более эффективным с помощью таблиц перекодировки.

Как повысить эффективность работы программы?

Тема эффективности, очень часто затрагиваемая, не так важна как многие склонны думать. Большая часть кода в большинстве программ не влияет на время исполнения. Если время, занимаемое каким-то участком кода, мало по сравнению с общим временем исполнения, то для этого участка гораздо важнее простота и мобильность, чем эффективность. (Помните, что компьютеры очень, очень быстры и даже «неэффективный» участок кода может выполняться без видимой задержки).

Печально известны попытки предсказать «горячие точки» программы.

Когда эффективность программы имеет значение, важно использовать профилировщики для определения тех участков программы, которые заслуживают внимания. Часто основное время выполнения поглощается периферийными операциями, такими как ввод/вывод и выделение памяти, которые можно ускорить с помощью буферизации и хеширования.

Для небольших участков программы, критичных в смысле эффективности, жизненно важно выбрать подходящий алгоритм; «микрооптимизация» этого участка менее важна. Многие часто предлагаемые «приемы по увеличению эффективности» (вроде замены операции сдвига умножением на степень двойки) выполняются автоматически даже неизощренными компиляторами.

Неуклюжие попытки оптимизации способны так увеличить размер программы, что ее эффективность упадет.

Правда ли, что применение указателей более эффективно, чем применение массивов? Насколько замедляет программу вызов функции? Быстрее ли ++i чем i = i + 1?

Точные ответы на эти и многие другие похожие вопросы, конечно же, зависят от процессора и применяемого компилятора. Если знать это необходимо, придется аккуратно определить время выполнения тестовых программ. (Часто различия столь незначительны, что потребуются сотни тысяч повторений, чтобы их увидеть. Если есть возможность, посмотрите ассемблерный листинг, выдаваемый компилятором, чтобы убедиться в различной трансляции двух претендующих на первенство альтернатив).

«Обычно» быстрее продвигаться по большим массивам с помощью указателей, чем с помощью индексов, однако есть процессоры, для которых справедливо обратное.

Хотя вызовы функций и увеличивают время выполнения, сами функции настолько повышают модульность и простоту понимания программы, что едва ли полезно от них отказываться.

Прежде чем переписывать выражения типа **i=i+1**, вспомните, что имеете дело с компилятором Си, а не с программируемым калькулятором. Любой приличный компилятор будет одинаково транслировать **++i**, **i+=1**; **i=i+1**.

Использовать **++i**, **i+=1** или **i=i+1** — вопрос стиля, а не эффективности.

Почему не выполняется такой фрагмент:
char *p = "Hello, world!";
p[0] = tolower(p[0]);

Стринговые константы не всегда можно модифицировать, за исключением случая, когда ими инициализируется массив. Попробуйте:

```
char a[] = "Hello, world!";
```

(Для компиляции старых программ некоторые компиляторы имеют ключ, который управляет возможностью модификации стринговых констант.)

Моя программа аварийно завершается еще до выполнения! (если использовать отладчик, то видно, что смерть наступает еще до выполнения первой инструкции в main)

Видимо, у вас один или несколько очень больших (более килобайта) локальных массивов. Во многих системах размер стека фиксирован, а операционные системы, в которых осуществляется динамическое выделение стековой памяти, (например, UNIX) могут быть введены в заблуждение, когда размер стека резко увеличивается.

Часто предпочтительнее объявить большие массивы типа **static** (если, конечно, каждый раз при рекурсивном вызове не требуется свежий массив).

Что означают сообщения «Segmentation violation» и «Bus error»?

Это значит, что программа пытается получить доступ к несуществующей или запрещенной для нее области памяти. Это постоянно происходит из-за неинициализированных или неверно инициализированных указателей, по вине **malloc** или, может быть, **scanf**.

Моя программа аварийно завершается, очевидно, при выполнении malloc, но я не вижу в ней ничего плохого

К несчастью, очень легко разрушить внутренние структуры данных, создаваемые **malloc**, а возникающие проблемы могут быть трудны для отладки. Чаще всего проблемы возникают при попытке записать больше данных, чем может уместиться в памяти, выделенной **malloc**; особенно распространена ошибка **malloc(strlen(s))** вместо **strlen(s) + 1**.

Другие проблемы включают освобождение указателей, полученных не в результате выполнения **malloc**, или попытки применить функцию **realloc** к нулевому указателю.

Существует несколько отладочных пакетов, чтобы помочь отследить возникающие при применении **malloc** проблемы.Есть у кого-нибудь комплект тестов для Си компилятора?

Где достать грамматику Си для программы YACC?

Самая надежная — конечно же грамматика из стандарта ANSI. Другая грамматика, подготовленная Джимом Роскиндом (Jim Roskind), находится на ics.uci.edu в директории pub/*grammar*. Одетый в плоть, работающий образец ANSI грамматики (принадлежащий Джефу Ли(Jeff Lee)) находится на

uunet в директории usenet/net.sources/ansi.c.grammar.Z (вместе с лексическим анализатором).

Мне необходим исходный текст для разбора и вычисления формул

Есть два доступных пакета — «**defunc**» и пакет «**parse**».

Мне необходима функция типа strcmp, но для приблизительного сравнения, чтобы проверить две строки на близость, но не на тождество

Обычно такие сравнения включают алгоритм «soundex», который ставит в соответствие сходно звучащим словам один и тот же числовой код. Этот алгоритм описан в томе «Сортировка и поиск» классической книги Дональда Кнута «Искусство программирования для ЭВМ».

Как по дате найти день недели?

Используйте **mktime** или попробуйте вот эту функцию:

```
dayofweek(y, m, d)  /* 0 = Воскресенье */
int y, m, d;        /* 1 <= m <= 12, y > 1752 (примерно) */
    {
    static int t[] = {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
    y -= m < 3;
    return (y + y/4 - y/100 + y/400 + t[m-1] + d) % 7;
    }
```

Как произносить «char»?

Ключевое слово Си «**char**» можно произносить тремя способами, как английские слова «**char**», «**care**» или «**car**». Выбор за вами.

Какие есть хорошие книги для изучения Си?

Митч Райт (Mitch Wright) поддерживает аннотированную библиографию книг по Си и по UNIX; она доступна через ftp ftp.rahul.net в директории pub/mitch/YABL.

Как преобразовать AnsiString в char*?

У класса AnsiString есть метод, декларация которого выглядит так:

```
char* __fastcall c_str() const;
   E.g.: char a[10];
  AnsiString b="CBuilder";
  strcpy(a, b.c_str());
```

Как сделать, чтобы программа на CBuilder3,4 не требовала .bpl, .dll?

В **Project ➪ Options ➪ Packages** снять галку с **Build with runtime packages**; в **Project ➪ Options ➪ Linker** снять галку с **Use dynamic RTL**.

Что такое RXLib и где его взять?

Одна из самых, если не самая лучшая библиотека общего назначения для Delphi. Огромное количество компонентов и полезных функций. Полные исходные тексты. Совместима со всеми Delphi, а также с C++Builder. Великолепные примеры использования. Исчерпывающие файлы помощи на русском языке.

Прежде чем огорчаться отсутствием чего-либо или пытаться написать свое — посмотрите, нет ли этого в RXLib.

Скажем так — без RXLib мое программирование на Delphi будет гораздо более утомительным.

Как сделать, чтобы окно вело себя, как верхняя панель в билдере, т.е. ресайзилось только по горизонтали, и только до определенного минимального размера, а по вертикали размер был фиксированным?

Надо написать обработчик сообщения **WM_GETMINMAXINFO**.

Например, так:

```
class TForm1 : public TForm
{
//...........
private:
   void __fastcall WMGetMinMaxInfo(TMessage& Msg);
BEGIN_MESSAGE_MAP
   VCL_MESSAGE_HANDLER(WM_GETMINMAXINFO, TMessage,
WMGetMinMaxInfo)
END_MESSAGE_MAP(TForm)
};
void __fastcall TForm1::WMGetMinMaxInfo(TMessage&Mmsg)
{
 (LPMINMAXINFO(Msg.LParam))->ptMinTrackSize.x=200;
 (LPMINMAXINFO(Msg.LParam))->ptMinTrackSize.y=Height;
 (LPMINMAXINFO(Msg.LParam))->ptMaxTrackSize.y=Height;
  Msg.Result=0;
}
```

В CB4 можно воспользоваться свойством **Constraints**.

Как организовать SplashScreen?

1. Посмотреть на $(BCB)\Examples\DBTasks\MastApp.

2. Воспользоваться функцией **ShowSplashWindow(...)** из RXLib.

3. Написать руками:

а) Делаешь форму, которая будет изображать **SplashScreen**;

б) Делаешь **WinMain** вида:

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
 {
    try
    {
 SplashF=new TSplashF(Application);
 SplashF->Show();
 SplashF->Update();

 Application->Initialize();
  //...

 SplashF->Close();
 delete SplashF;

 Application->Run();
  //...
```

Как засунуть иконку в system tray («туда, где часы» (с))?

1. Воспользоваться компонентом **TRxTrayIcon** из **RXLib**.

2. Посмотреть в хелпе описание на **Shell_NotifyIcon(...)**.

3. Посмотреть на $(BCB)\Examples\Apps\TrayIcon (есть только в CB3,4).

4. Посмотреть на $(BCB)\Examples\Controls\Tray (CB4).

Как русифицировать Database Desktop 7?

```
[HKEY_CURRENT_USER\Software\Borland\DBD\7.0\Preferences\
Properties]
     "SystemFont"="MS Sans Serif"
```

или

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\
CodePage]
```

```
"1252"="c_1251.nls"
```

И все!!! Никаких проблем с «иероглифами» в любых программах!

Из-за чего может виснуть C++Builder 3 под Windows 98 (при запуске)? Он запускался в Windows 95 при 16 цветах, а в Windows 98 никак не хочет

Надо либо убавлять **Hardware Acceleration**, либо менять драйверы.

```
[HKEY_CURRENT_CONFIG\Display\Settings]
       "BusThrottle"="on"
```

Почему в билдере размер структуры всегда растягивается до кратного 4-ем?

Из-за выравнивания (RTFM Data Alignment).

Чтобы поля структуры выравнивались на 8-ми битную границу, необходимо использовать следующую конструкцию:

```
#pragma pack(push, 1)
<structure definition>
#pragma pack(pop)
```

Менять выравнивание для всего проекта (**Project Options\Advanced Compiler\Data Alignment**) не рекомендуется.

Какой-нибудь из CBuilder'ов умеет делать win16 Exe?

Нет.

Как создать компонент по его имени?

```
#include <typeinfo.h>
#include <stdio.h>
class A {
  public:
     virtual A *Create(void) = 0;
};

class B1 : A {
  public:
     B1();
     A *Create(void) { return(new B1); }
};

class B2 : A {
  public:
     B2();
```

```
     A *Create(void) { return(new B2); }
};

B1::B1() { printf("Create B1\n"); }
B2::B2() { printf("Create B2\n"); }

// Собственно "создатель"

A *CopyCreate(A *a)
{
  if(a && typeid(A).before(typeid(a))) return(a->Create());
  else printf("Illegal call\n");
  return(NULL);
}

// дальше пример использования

void main( void )
{
  B1 *b1 = new B1;
  B2 *b2 = new B2;

 printf("Call test b1\n");
 B1 *bb1 =
dynamic_cast<B1*>(CopyCreate(reinterpret_cast<A*>(b1)));
 printf("Call test b2\n");
 B2 *bb2 =
dynamic_cast<B2*>(CopyCreate(reinterpret_cast<A*>(b2)));

 delete b;
 delete bb2;
 delete b1;
 delete b2;
}
--------------------результат запуска-----------
G:\PROJECT.BC5\Test>a.exe
Create B1
Create B2
Call test b1
Create B1
Call test b2
```

```
Create B2
--------------------------------------------------
```

Естественно для «полной культурности» надо понавставлять **try/catch** или перекрыть **Bad_Cast**, но это уже детали.

Еще один способ:

```
class TComponent1* : public TComponent
{
// Это класс от которого мы будем порождать все наши классы
public:
  __fastcall TComponent1( TComponent* Owner
):TComponent(Owner){}

  virtual TComponent1* __fastcall Create(TComponent*
Owner)=0;
}

class TMyClass1 : public TComponent1
{
public:
  __fastcall TMyClass1(TComponent* Owner):
TComponent1(Owner){}
  virtual TMyClass1* __fastcall Create(TComponent* Owner)
        {return new TMyClass1(Owner);}
// Эта функция создает класс, поскольку все создаваемые
классы мои и порождены от TObject проблем нет, осталось
только ее вызвать.
}
```

Вот функция для создания класса:

```
TComponent1* __fastcall CreateClass( AnsiString ClsName,
TComponent* Owner )
{
   TClass cls = GetClass( clsName );
// Это сработает если класс зарегистрирован функцией
RegisterClasses в инициализации модуля
   void * mem = SysGetMem( InstanceSize(cls) );
   TComponent1* Result = InitInstance(cls, mem);
   Result = Result->Create( Owner );
// Класс создан правильно и его можно вернуть освободив
память
```

```
        SysFreeMem( mem );
        return Result;
    }
```

Если список классов, которые надо создавать по имени, не очень велик, то можно так:

```
TControl* CreateControlByName(AnsiString ClassName,
TComponent *Owner)
   {
      TMetaClass *c=GetClass(ClassName);
      if(c==NULL)
  throw Exception("Unregistered class.");
      if(c==__classid(TButton))
  return new TButton(Owner);
      if(c==__classid(TEdit))
  return new TEdit(Owner);
      return NULL;
   }
```

Почему функция isdigit (да и остальные is*) возвращает некорректные значения для аргумента в виде русской буквы?

Напиши **#undef isdigit**, будет вызываться функция с правильным кастингом.

Почему при сборке в CB3 с включенным Build With Runtime Packages все работает, а если отключить, то вылетает с ошибкой, не доходя до Application->Initialize()?

В IDE есть глючек, в результате которого порядок **.lib** в строке **LIBRARIES .bpr**-файла оказывается неправильным (первым обязательно должен идти **vcl35.lib**). Из-за этого нарушается порядок инициализации модулей и глобальных VCL-объектов. В результате при запуске программы имеем стабильный **Access Violation**. Для его устранения необходимо поправить строку **ALLLIB .bpr**-файла:

```
ALLLIB = vcl35.lib $(LIBFILES) $(LIBRARIES) import32.lib
cp32mt.lib
```

Есть функция, которая производит длительные вычисления в цикле. Хотелось бы иметь возможность ее прервать. Естественно, что пока вычисления не выходят из цикла никакие контролы не работают…

Вставить в цикл, в котором происходят вычисления, вызов **Application->ProcessMessages()**; Т.е.:

```
   for(.....
   {
```

```
// здесь выполняются вычисления
    Application->ProcessMessages();
}
```

Также вынести вычисления в отдельный **thread**.

Я переписываю BDE-приложение на другой компьютер, а оно отказывается работать. Что делать?

1. Использовать инсталляционный пакет, например **InstallShield** или **Wise**.

2. Не использовать его. В этом случае нет универсального решения.

Оно будет варьироваться в зависимости от использования BDE в локальном или серверном режиме, для доступа к Paradox- или DBF-таблицам, использования локального SQL, версии BDE, и так далее... Здесь приведен пример для наиболее общего варианта — пятая версия BDE, локальные таблицы, без использования локального SQL, стандартная кодировка ANSI.

Нужно добавить следующие файлы из папки BDE к вашему исполняемому модулю: blw32.dll, idapi32.dll, idr20009.dll, idpdx32.dll для Paradox-таблиц или iddbas32.dll для DBF-таблиц, bantam.dll, charset.cvb, usa.btl.

Доступ к таблицам надо настроить не через псевдонимы (alias'ы), а через пути в файловой системе. В идеале все таблицы храните в папке программы, тогда нужно только указать имя таблицы без пути.

Приготовленный таким образом дистрибутив запускается на любой машине без необходимости инсталляции BDE, максимально устойчив и нечувствителен к смене имен папок/переинсталляции системы/порчи реестра/влиянии на другие BDE-приложения. Добавка к основному модулю составляет для этих семи dll-библиотек ~1030 КБ, после упаковки ~470 КБ.

Для того, чтобы установить программу, которая требует BDE, есть несколько базовых путей, в частности:

1. Создать полноценную программу инсталляции с помощью продуктов Install Shield, Wise или подобных. Указанные продукты используются чаще всего и оба позволяют включить в инсталляцию BDE + базовые настройки (алиасы и пути).

2. Для разных целей можно сделать инсталляцию BDE отдельным пакетом (в Install Shield'e это делается более чем элементарно — в проект не надо добавлять ничего, кроме поддержки BDE). Удобно в процессе написания программы для одного пользователя. Первый раз устанавливаешь и настраиваешь BDE, а затем носишь только новые версии программ. Так же можно при установке Дельфи/Билдера с компашки снять флажки отовсюду кроме BDE — в этом случае будет установлена только BDE.

3. Есть возможность инсталлировать BDE ручками. Первый этап — копирование файлов, второй — прописывание реестра.

Теперь к вопросу о том, почему установка BDE — это не просто прописать одну опцию в проекте.

Дело в том, что BDE — это не просто несколько библиотек динамического доступа (DLL), это — целый **engine**, достаточно хорошо продуманный для того, чтобы быть и универсальным и расширяемым. Занимает он в запакованном виде две дискеты, а в распакованном (+ файлы, которые включать в поставку не нужно) — более десяти!

Естественно, не для всех задач подходит именно BDE (благодаря своим особенностям). Во-первых, возникают проблемы при работе с DBF форматов Clipper и Fox. Во-вторых, не для всех программ требуются все возможности BDE, а быть они должны как можно меньше.

Как из Builder'a можно работать с последовательными портами?

Существует компонент ZComm (free for personal use), поддерживает все порты, все скорости, **hard/soft flow control**, **in/out** буферизацию. Передача/прием данных вынесены в отдельную нитку.

Еще один вариант:

```
__fastcall TComPort::TComPort(TComponent* Owner) :
TComponent(Owner)
{
 OverlappedStructure.Offset      = 0;
 OverlappedStructure.OffsetHigh = 0;
 OverlappedStructure.hEvent      = 0;
 iComNumber = 2;
 iBaudRate = 9600;
```

```
 hCom = INVALID_HANDLE_VALUE;
}
//---------------------------------------------------------
int __fastcall TComPort::Open(int n)
{
 bool ierr;
 AnsiString ComName;
 ComName = "\\\\.\\COM"+IntToStr(n);

 if(hCom != INVALID_HANDLE_VALUE) Close();

 hCom = CreateFile(ComName.c_str(),
GENERIC_READ|GENERIC_WRITE, 0, 0,

  OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED,
0);
 if (hCom == INVALID_HANDLE_VALUE)
   throw Exception
("Невозможно открыть порт COM"+IntToStr(n));
 SetupComm(hCom, 2048, 2048);

 GetCommTimeouts(hCom, &Timeouts);
 Timeouts.ReadIntervalTimeout = MAXDWORD;
 Timeouts.ReadTotalTimeoutMultiplier = 0;
 Timeouts.ReadTotalTimeoutConstant = 0;
 Timeouts.WriteTotalTimeoutMultiplier = 0;
 Timeouts.WriteTotalTimeoutConstant = 0;
 ierr = SetCommTimeouts(hCom, &Timeouts);

 if(!ierr) throw Exception
("Ошибка инициализации порта COM"+IntToStr(n));

 GetCommState(hCom, &dcbBuf);
 dcbBuf.BaudRate = iBaudRate;
 dcbBuf.fBinary = true;
 dcbBuf.fParity = false;
 dcbBuf.ByteSize = 8;
 dcbBuf.Parity = 0;
 dcbBuf.StopBits = 0;
 ierr = SetCommState(hCom, &dcbBuf);
```

```
 if(!ierr) throw Exception
("Ошибка инициализации порта COM"+IntToStr(n));

 ierr = SetCommMask(hCom, EV_RXCHAR);

 if(!ierr) throw Exception
("Ошибка инициализации порта COM"+IntToStr(n));

 return iComNumber = n;
}
//--------------------------------------------------
int __fastcall TComPort::Open(void)
{
 return Open(iComNumber);
}
//-------------------------------------------------
void __fastcall TComPort::Close(void)
{
 CloseHandle(hCom);
 hCom = INVALID_HANDLE_VALUE;
}
//-------------------------------------------------
void __fastcall TComPort::FlushBuffers(void)
{
 PurgeComm(hCom,
PURGE_TXABORT|PURGE_TXCLEAR|PURGE_RXABORT|PURGE_RXCLEAR);
}
//--------------------------------------------------
DWORD __fastcall TComPort::WriteBlock(void *buf, int count)
{
 DWORD realCount;
 WriteFile(hCom, buf, count, &realCount,
&OverlappedStructure);
 return realCount;
}
//--------------------------------------------------
DWORD __fastcall TComPort::ReadBlock(void *buf, int count)
{
 DWORD realCount;
 bool bResult = ReadFile(hCom, buf, count, &realCount,
&OverlappedStructure);
```

```
 // if there was a problem, or the async. operation's still
pending ...
 if(!bResult)
    {
     // deal with the error code
     switch(GetLastError())
      {
       case ERROR_HANDLE_EOF:
        {
 // we're reached the end of the file
 // during the call to ReadFile
 // code to handle that
 throw Exception("1");
        }
       case ERROR_IO_PENDING:
        {
 // asynchronous i/o is still in progress
 // do something else for a while
 Sleep(100);

 // check on the results of the asynchronous read
 bResult = GetOverlappedResult(hCom, &OverlappedStructure,
&realCount,
  false);

 // if there was a problem ...
 if(!bResult)
    {
     // deal with the error code
     switch(GetLastError())
      {
       case ERROR_HANDLE_EOF:
        {
 // we're reached the end of the file
 //during asynchronous operation
 throw Exception("2");
        }
        // deal with other error cases
       default:
        {
 throw Exception("3");
```

```
          }
        }
      }
          } // end case

        // deal with other error cases
        default:
          {
 throw Exception("4");
          }

      } // end switch
    } // end if

 return realCount;
}
//---------------------------------------------------
void __fastcall TComPort::SetBaudRate(int b)
{
 GetCommState(hCom, &dcbBuf);
 dcbBuf.BaudRate = b;
 SetCommState(hCom, &dcbBuf);
}
//----------------------------------------------------
DWORD __fastcall TComPort::ClearError(void)
{
 COMSTAT stCom;
 DWORD ierr;
 ClearCommError(hCom,&ierr,&stCom);
 return ierr;
}
```

Как отследить запуск второй копии приложения?

1. Воспользоваться функцией **FindWindow()**. Ее использование затруднительно если меняется заголовок окна или есть другое окно с таким же заголовком и классом окна.

2. Воспользоваться RxLib-овской функцией **ActivatePrevInstance**, которая в конце-концов тоже использует эту функцию. Однако **ActivatePrevInstance** так же выполняет некоторые полезные действия (активизация предыдущей копии приложения).

3. Можно создавать семафоры, мутексы, но тогда при некорректном завершении программы, ты ее больше не запустишь.

Пример использования мутекса:

```
HANDLE hMutex=CreateMutex(NULL, FALSE, "YourMutexName");
    if(GetlastError()==ERROR_ALREADY_EXISTS )
    {
// здесь надо бы активизировать предыдущую копию приложения.
// как это сделать, см. ActivatePrevInstance().
    }
    else
    {
 try
 {
     Application->Initialize();
     Application->CreateForm(__classid(TForm1), &Form1);
     Application->Run();
 }
 catch (Exception &exception)
 {
    Application->ShowException(&exception);
 }
 CloseHandle(hMutex);
    }
```

4. Можно получить имя исполняемого файла для каждого из запущенных процессов, после чего сравнить его с именем **.exe** вашего процесса... Недостатки способа:

а) Две копии приложения могут быть запущены из разных мест.

б) Различные методы получения списков запущенных процессов для '9x и NT.

Пример для '9x:

```
#include <tlhelp32.h>
#include <dos.h>

USERES("Project1.res");
USEFORM("Unit1.cpp", Form1);
//---------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
```

```
{
  HANDLE hSnapshot=CreateToolhelp32Snapshot
(TH32CS_SNAPPROCESS, 0);
  PROCESSENTRY32 pe;
  pe.dwSize=sizeof(pe);
  bool Running=false;
  DWORD CurrentProc=GetCurrentProcessId();
  if(Process32First(hSnapshot, &pe))
    do
    {
      if(CurrentProc!=pe.th32ProcessID &&
strcmpi(pe.szExeFile, _argv[0])==0)
      {
 Running=true;
 break;
      }
    }while(Process32Next(hSnapshot, &pe));

  CloseHandle(hSnapshot);

  if(Running)
    return 1;

  try
  {
    Application->Initialize();
//......
```

5. Использовать временный файл:

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  HANDLE hFile = CreateFile("c:\\tempfile.tmp",
GENERIC_WRITE, 0,
          NULL, CREATE_ALWAYS,
          FILE_ATTRIBUTE_NORMAL | FILE_FLAG_DELETE_ON_CLOSE,
          NULL);
  if(hFile == INVALID_HANDLE_VALUE)
      return 1;
  try
  {
    Application->Initialize();
```

```
    Application->CreateForm(__classid(TForm1), &Form1);
    Application->Run();
  }
  catch (Exception &exception)
  {
    Application->ShowException(&exception);
  }

  CloseHandle(hFile);

  return 0;
}
```

Это, в принципе, универсальный способ, устойчивый к некорректному завершению программы, основным недостатком которого является появление «лишнего» файла на диске.

Как на C++ выглядит паскалевский is?

```
dynamic_cast<...>(...);
```

Пример:

Паскаль:

```
if Screen.Forms[I] is FormClass then begin
```

C++:

```
if (dynamic_cast<FormClass*>(Screen->Forms[I])){
```

Как сделать окно как у WinAMP?

Установки формы:

```
= Object Inspector =
BorderIcons=[]
BorderStyle=bsNone
```

Если таскаем за **TLabel**, то поместить на форму один **Label** и 3 кнопки **SpeedButton** (свернуть, развернуть, закрыть), в процедуре на событие **onMouseDown** поместить следующие строчки:

```
void __fastcall TForm1::Label1MouseDown(TObject *Sender,
TMouseButton Button,
     TShiftState Shift, int X, int Y)
{
  const int SC_DRAGMOVE = 0xF012;
  if(WindowState!=wsMaximized)
// чтобы не таскать развернутое окно
```

```
  {
  ReleaseCapture();
  Perform(WM_SYSCOMMAND, SC_DRAGMOVE, 0);
  }
}

// на кнопки в событии onClick

// свертывание формы

void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
{
Perform(WM_SYSCOMMAND,SC_MINIMIZE,0);
}

// развертывание/восстановление формы

void __fastcall TForm1::SpeedButton2Click(TObject *Sender)
{
if(WindowState==wsMaximized)
//тут не плохо бы сменить рисунок на кнопке
  Perform(WM_SYSCOMMAND,SC_RESTORE,0);
else
  Perform(WM_SYSCOMMAND,SC_MAXIMIZE,0);
}

// закрытие формы

void __fastcall TForm1::SpeedButton3Click(TObject *Sender)
{
Perform(WM_SYSCOMMAND,SC_CLOSE,0);
}
```

Все объекты могут находиться на панели (TPanel) — но проще поместить **Bevel** на форму.

Почему не работает код:
Variant v = Variant::CreateObject("Excel.Application");
v.OlePropertySet("Visible",true);

Из-за особенностей реализации OLE-сервера Excel русской локализации.

В Borland`s examples сказано, что примеры с OLE работают, только если у вас стоит английская версия Word или Excel.

Необходимо использовать библиотеку типов Excel.

Как показать ProgressBar на StatusBar'e?

Предположим, что вы хотите показать **CProgressCtrl** на весь **StatusBar**.

Для этого необходимо проделать следующее:

- Выберите пункт меню **View ➪ Resource Symbols**. Нажмите кнопку **New** и добавьте новое имя, в нашем примере это будет **ID_PROGRBAR**.
- В файле **MainFrm.cpp** найдите объявление массива **indicators** (он находится сразу после **END_MESSAGE_MAP**) и отредактируйте его к следующему виду:

```
static UINT indicators[] =
{
ID_PROGRBAR
};
```

- В файле **MainFrm.h** создайте **protected** переменную **m_bCreated** типа **BOOL** и **public** переменную **m_progress** типа **CProgressCtl**.
- В файле **MainFrm.cpp** отредактируйте конец функции **int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)** таким образом:

```
 if (!m_wndStatusBar.Create(this ) ||
 !m_wndStatusBar.SetIndicators(indicators,
 sizeof(indicators)/sizeof (UINT)))
    {
  TRACE0("Failed to create status bar\n" );
  return -1; // fail to create
     }
```

добавьте следующую строку:

```
   else {
m_wndStatusBar.SetPaneInfo(0,ID_PROGRBAR,SBPS_STRETCH,10);
 }
```

Кроме того, добавьте инициализацию нашей переменной **m_bCreated**:

```
.........
m_bCreated=FALSE;
..........
```

- Теперь мы можем использовать **ProgressBar** в строке статуса, естественно не забыв создать этот объект. Предположим, у нас есть функция **CMainFrame::OnWork()**. Она будет выглядеть примерно так:

```
 void CMainFrame::OnWork()
      {
        RECT rc;
  m_wndStatusBar.GetItemRect(0,&rc);
 if (m_bCreated==FALSE)
 {
   // создаем m_progress
   m_progress.Create(WS_VISIBLE|WS_CHILD, rc,&m_wndStatusBar,
1);
// Устанавливаем размер от 0 до 100
   m_progress.SetRange(0,100);
   m_progress.SetStep(1);
   m_bCreated=TRUE;
  }
  for (int I = 0; I < 100; I++)
  {
    Sleep(20);
    m_progress.StepIt();
  }
      }
```

- Если откомпилировать проект на этой фазе, то все будет работать, но при изменении размера окна линейка ProgressBar'а размеры менять не будет, поэтому необходимо перекрыть событие **OnSize**:

```
      void CMainFrame::OnSize(UINT nType, int cx, int cy)
      {
  CFrameWnd::OnSize(nType, cx, cy);
   if (m_bCreated)
     {
 RECT rc;
 m_wndStatusBar.GetItemRect(0,&rc);
 m_progress.SetWindowPos(&wndTop, rc.left, rc.top,
```

```
rc.right - rc.left,rc.bottom - rc.top, 0);
     }
      }
```

- Вот теперь все. Откомпилируйте проект и убедитесь, что все работает.

Как использовать CTreeCtrl для построения дерева каталогов диска, как в Проводнике? Неужели необходимо рекурсивно просмотреть диск, а потом прописать ручками все Итемы данного контрола??

Это тормозно и глючно. На больших дисках это займет несколько минут. Если каталоги добавляются или удаляются другими приложениями во время работы твоего контрола, то будешь весь в проблемах. Все гораздо проще. Никаких рекурсий.

Просматриваем корневой каталог на предмет наличия подкаталогов и создаем итемы первого уровня, в которых создаем по одному фиктивному итему (чтобы крестик был и итем можно было раскрыть).

```
+ Каталог 1
+ Каталог 2
+ Каталог 3
```

Как только юзер пытается раскрыть итем, соответствующий некому каталогу, мы удаляем из него фиктивный итем, просматриваем этот подкаталог и добавляем соответствующие итемы со своими фиктивными внутри.

```
-Каталог 1
   + Каталог 4
   + Каталог 5
   + Каталог 6
+ Каталог 2
+ Каталог 3
```

Как только юзер закрывает итем, мы удаляем из него все дочерние итемы и обратно добавляем фиктивный. Если структура каталогов изменилась, для обновления юзеру достаточно просто закрыть и открыть соответствующую ветку.

Именно так и работает «Проводник».

Есть класс — потомок CListView. Как изменить стиль у объекта CListCtrl, принадлежащего к этому *view (например установить стиль Report)?

Для этого пишите в **OnInitialUpdate** вашего вида:

```
void CMyListView::OnInitialUpdate()
{
```

```
 ......
 CListView::OnInitialUpdate();

    CListCtrl& theCtrl = GetListCtrl();
    DWORD dwStyle=GetWindowLong(theCtrl.m_hWnd,GWL_STYLE);

SetWindowLong(theCtrl.m_hWnd,GWL_STYLE,dwStyle|LVS_REPORT);
....
```

Гораздо проще перекрыть **PreCreateWindow** (лучше всего воспользоваться ClassWizard-ом) и поковырять переданный по ссылке **CREATESTRUCT** типа такого:

```
BOOL CMyListView::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style|=LVS_REPORT;//так мы добавляем стиль
    cs.style&=LVS_REPORT;//а вот так снимаем

    return CMyListView::PreCreateWindow(cs);
}
```

Как CString привести к char *?

```
#include <atlbase.h>
USES_CONVERSION;
 CString strData(_T("Some Data"));
 char* lpszString = T2A((LPTSTR)(LPCTSTR)strData);
```

или

```
CString tmp_str;
char* st;
st=tmp_str.GetBuffer(tmp_str.GetLength())
```

важно то, что если с **tmp_str** что-либо сделать, то необходимо опять получить указатель на внутренний буфер **CString**.

Какие библиотеки Freeware/Commercial существуют для Visual C++?

1. BCG Control Library (freeware)

2. CJLibrary (freeware)

Фирма Stringray Software производит библиотеки для Visual C++ (MFC, ATL):

1. Stingray Objective Toolkit (PRO) — набор различных компонентов для MFC и ATL.

2. Stingray Objective Grid (PRO) — мощная сетка данных с возможностями, близкими к Excel. Дружит с базами данных (через DAO,ADO,ODBC). Можно использовать для ввода данных в таблицы БД и для вывода/печати простых отчётов.

3. Stingray Objective Chart — средство для построения диаграмм.

4. Stingray Objective Views — средство для создания Visio-подобных интерфейсов (при помощи векторной графики).

5. Stingray Objective Edit — текстовый редактор с подсветкой синтаксиса.

Кроме этих, есть и другие продукты.

Фирма Dundas Software производит библиотеки для Visual C++ (MFC):

1. Dundas Ultimate Toolbox — набор компонентов для MFC, по составу несколько отличающийся от Stingray Objective Toolkit.

2. Dundas Ultimate Grid — сетка данных, конкурент Stingray Objective Grid.

3. Dundas TCP/IP — реализация протоколов POP3, NEWS и т.п.

4. Dundas Chart — диаграммы и другие продукты.

А можно пример консольной программы?

```
#include <windows.h>
#include <stdlib.h>

void main()
 {
 HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
 SMALL_RECT srct;
 CHAR_INFO chiBuffer[160];
 COORD coord1, coord2;
 char ddd[666];
 CharToOem("2:5095/38 - злобный ламерюга", ddd);
 DWORD cWritten;
 coord1.Y = 0; coord1.X = 0;
 hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
 WriteConsoleOutputCharacter(hStdout, ddd, lstrlen(ddd),
coord1, cWritten);
```

```
 for (int i = 0; i {
 WORD wColors = 1 + i * 3;
 coord1.X = i;
 WriteConsoleOutputAttribute(hStdout, , 1, coord1, cWritten);
 }
 srct.Top = 0; srct.Left = 0; srct.Bottom = 1;
srct.Right = 79;
 coord1.Y = 0; coord1.X = 0;
 coord2.Y = 1; coord2.X = 80;
 ReadConsoleOutput(hStdout, chiBuffer, coord2, coord1, );
 for (i = 0; i {
 srct.Left = (SHORT)((double)(79 - lstrlen(ddd)) * rand() /
RAND_MAX);
 srct.Top = (SHORT)((double)25 * rand() / RAND_MAX);
 srct.Bottom = srct.Top + 1;
 WriteConsoleOutput(hStdout, chiBuffer, coord2, coord1, );
 }
```

Пытаюсь из своей программы вызвать Word97, для это делаю несколько импортов и в результате имею кучу ошибок. Как правильно?

```
// Office.h

#define Uses_MSO2000_

 #ifdef Uses_MSO2000
 // for Office 2000
 #import <mso9.dll>
 #import <vbe6ext.olb>
 #import <msword9.olb> rename("ExitWindows","_ExitWindows")
 #import <excel9.olb> rename("DialogBox","_DialogBox") \
 rename("RGB","_RGB") \
 exclude("IFont","IPicture")
 #import <dao360.dll> rename("EOF","EndOfFile")
rename("BOF","BegOfFile")
 #import <msacc9.olb>

#else
 // for Office 97
 #import <mso97.dll>
 #import <vbeext1.olb>
 #import <msword8.olb> rename("ExitWindows","_ExitWindows")
 #import <excel8.olb> rename("DialogBox","_DialogBox") \
```

```
 rename("RGB","_RGB") \
 exclude("IFont","IPicture")
 #import <DAO350.DLL> \
 rename("EOF","EndOfFile") rename("BOF","BegOfFile")
 #import <msacc8.olb>
#endif
```

Как отредактировать ресурсы .exe файла?

Это возможно лишь под NT.

Как программно получить номер билда своего приложения в VC++?

Штатной возможности нет, поскольку не все одинаково трактуют понятие «номер билда» и не все одинаково его используют. Однако большинство людей используют для хранения номера билда конкретного файла ресурсы типа **VERSIONINFO**, откуда эту информацию можно потом получить (для отображения в диалоге «О программе») с помощью функций из version.dll.

Упрощенно говоря, информация о версии файла хранится в **VERSIONINFO** в виде четырех чисел, значимость которых убывает слева направо. Например, для mfc42.dll из поставки Win2k версия файла выглядит как 6.0.8665.0. Здесь первая цифра совпадает с версией продукта (MSVC 6), вторая означает подверсию (MSVC 6.0), третья — номер билда. В своих dll-ках и exe-шниках Microsoft постоянно использует эту схему.

Обычно для автоматического увеличения номера версии используются макросы Visual Studio (== скрипты на VBScript), ковыряющие файл ресурсов проекта. Эти макросы либо связываются с кнопкой на тулбаре MSDev, либо вызываются из обработчика события **Application_BeforeBuildStart** в файле макросов. Исходник, реализующий номер билда, приведен ниже (должен работать на MSVC6SP3):

```
 Sub IncVersion()
 'DESCRIPTION: Increments file version
      Dim oDoc
      Dim iVer

      Set oDoc = Documents.Open
(Application.ActiveProject &".rc", "Text")
      if oDoc Is Nothing Then
   Exit Sub
```

```
            End If

            oDoc.Selection.FindText "FILEVERSION", dsMatchCase
            if Len(oDoc.Selection) = 0 Then
        oDoc.Close dsSaveChangesNo
        Set oDoc = Nothing
        Exit Sub
            End If
            oDoc.Selection.EndOfLine
            oDoc.Selection.FindText ",", dsMatchBackward
            oDoc.Selection.CharLeft
            oDoc.Selection.WordLeft dsExtend
            iVer = oDoc.Selection
            iVer = iVer + 1
            oDoc.Selection = iVer

            oDoc.Selection.FindText """FileVersion""", dsMatchCase
            if Len(oDoc.Selection) = 0 Then
        oDoc.Close dsSaveChangesNo
        Set oDoc = Nothing
        Exit Sub
            End If
            oDoc.Selection.EndOfLine
            oDoc.Selection.FindText ",", dsMatchBackward
            oDoc.Selection.CharLeft
            oDoc.Selection.WordLeft dsExtend
            iVer = oDoc.Selection
            iVer = iVer + 1
            oDoc.Selection = iVer

            oDoc.Close dsSaveChangesYes
            Set oDoc = Nothing

    End Sub
```

Какой функцией можно переключить видеорежим?

Этим занимается **ChangeDisplaySettings(...);**

Вот пример, который устанавливает разрешение 640x480 (24 bit):

```
=== Cut ===
    DEVMODE md;
```

```
    ZeroMemory(&md, sizeof(md));
    md.dmSize = sizeof(md);
    md.dmFields = DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;
    md.dmBitsPerPel = 24;
    md.dmPelsWidth = 640;
    md.dmPelsHeight = 480;
    ChangeDisplaySettings(&md, 0);
=== Cut ===
```

Как вызвать окно выбора папки?

Воспользуйтесь следующей функцией:

```
BOOL FGetDirectory(LPTSTR szDir)
{ BOOL fRet;
    TCHAR szPath[MAX_PATH];
    LPITEMIDLIST pidl;
    LPITEMIDLIST pidlRoot;
    LPMALLOC lpMalloc;
    BROWSEINFO bi =
         {
         NULL,
         NULL,
         szPath,
         "Выберите папку",
         BIF_RETURNONLYFSDIRS,
         NULL,
         0L,
         0
        };
    if (0 != SHGetSpecialFolderLocation(HWND_DESKTOP,
CSIDL_DRIVES, &pidlRoot))
       return FALSE;
    if (NULL == pidlRoot)
       return FALSE;
    bi.pidlRoot = pidlRoot;
    pidl = SHBrowseForFolder(&bi);
    if (NULL != pidl)
       fRet = SHGetPathFromIDList(pidl, szDir);
    else
       fRet = FALSE; // Get the shell's allocator to free
PIDLs
    if (!SHGetMalloc(&lpMalloc) && (NULL != lpMalloc))
```

```
    {
      if (NULL != pidlRoot)
      {
        lpMalloc->Free(pidlRoot);
      }
      if (NULL != pidl)
      {
        lpMalloc->Free(pidl);
      }
lpMalloc->Release();
}
return fRet;
}

LPTSTR PszAlloc(int cch)
{
return (LPTSTR) LocalAlloc(LMEM_FIXED, sizeof(TCHAR) *
(cch+1));
}

bool PszDeAlloc(HLOCAL mem_ptr)
{
return (LocalFree(mem_ptr)==NULL) ? true : false;
}
```

Затем, при необходимости предложить пользователю выбрать папку используйте примерно такой код:

```
....
LPTSTR fname;
fname=PszAlloc(250);
FGetDirectory(fname);
......
PszDeAlloc((HLOCAL)fname);
```

Приложения

Средства для разработчиков

Каталог средств для разработчиков, программирующих на языках C/C++

http://www.progsource.com/ c_development.html

Inprise Borland C++

Узел, посвященный компилятору Inprise Borland C++.

http://www.inprise.com/borlandcpp/

IBM VisualAge for C++

Узел, посвященный компилятору IBM VisualAge for C++.

http://www.software.ibm.com/ad/ visualage_c++/

Inprise C++Builder

Узел, посвященный компилятору Inprise C++Builder.

http://www.inprise.com/bcppbuilder/

Metrowerks CodeWarrior

Узел, посвященный CodeWarrior фирмы Metrowerks.

http://www.metrowerks.com/

Powersoft Power++

Узел, посвященный компилятору Powersoft Power++ фирмы Powersoft/Sybase.

http://www.sybase.com/products/powerpp/

Symantec C++

Узел, посвященный компилятору Symantec C++.

http://www.symantec.com/scpp/index_product.html

Watcom C/C++

Узел, посвященный компилятору Watcom C/C++ фирмы Powersoft/Sybase.

http://www.sybase.com/products/languages/watccpl.html

MetaWare High C++

Узел, посвященный компилятору High C++ фирмы MetaWare.

http://www.metaware.com/techno/ techno.html

Visual C++

Узел, посвященный компилятору Visual C++ фирмы Microsoft.

http://msdn.microsoft.com/visualc/

Базовые алгоритмы на C++

Примеры реализации различных алгоритмов на языке C++.

http://people.we.mediaone.net/stanlipp/ generic.html

Ссылки на ресурсы по C++

Различные ссылки на ресурсы по C++.

http://www.enteract.com/~bradapp/links/cplusplus-links.html

Ссылки на ресурсы по C++

Множество ссылок на различные ресурсы по C++.

http://webnz.com/robert/cpp_site.html

Ссылки на ресурсы по C++

Множество ссылок на различные ресурсы по **C++**.

http://www.cs.bham.ac.uk/~jdm/cpp.html

Ссылки на ресурсы по C++

Множество ссылок на различные ресурсы по **C++**.

http://www.kfa-juelich.de/zam/cxx/extern.html

Стандарт языка C++ (1997)

Онлайновая версия стандарта языка C++ 1997 года.

http://www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/index.html

C++ Syntax

Краткий справочник по синтаксису языка C++.

http://www.csci.csusb.edu/dick/c++std/ syntax.html

Коллекция ссылок по C++

Множество ссылок на различные ресурсы по **C++**.

http://webopedia.internet.com/TERM/C/ C_plus_plus.html

Коллекция ссылок по C++

Множество ссылок на различные ресурсы по C++.

http://walden.mo.net/~mikemac/clink.html

Коллекция ссылок по C++

Множество ссылок на различные ресурсы по C++.

http://www.austinlinks.com/CPlusPlus/

Rogue Wave Software

Различные библиотеки классов — Analytics.h++, DBTools.h++, Money.h++, Standard C++ Library, Threads.h++, Tools.h++ и Tools.h++ Professional.

http://www.roguewave.com/products/ products.html

Powersoft Power++ Resources

Ресурсы для пользователей компилятора Powersoft Power++.

http://www.orbit.org/power/resources.html

The C++ Builder Programmer's Ring

Ссылки на Web-узлы, входящие в The C++ Builder Programmer's Ring.

http://www.webring.org/cgi-bin/webring? ring=cbuilder&list

The C++Builder Web Ring

Ссылки на узлы, входящие в The C++Builder Web Ring.

http://members.aol.com/zbuilder/ resource.htm

Association of C and C++ Users

Web-узел ассоциации пользователей языков Си и C++ (ACCU).

http://www.accu.org/

ObjectSpace C++ Toolkit

Библиотеки классов ObjectSpace C++ Toolkit фирмы ObjectSpace: Communications ToolKit, Foundations ToolKit, Internet ToolKit, Standards ToolKit, STL ToolKit, Systems ToolKit, Web ToolKit.

http://www.objectspace.com/toolkits/

C++Builder Sites

Множество ссылок на различные ресурсы по C++Builder.

http://www.cbuilder.dthomas.co.uk/resources/bcbsites.htm

Standard Template Library

Руководство программиста по библиотеке Standard Template Library (STL).

http://www.sgi.com/Technology/STL/

Примеры кода

Различные примеры кода на C++.

http://www.geocities.com/SiliconValley/ Bay/1055/snippets.htm

Примеры кода

Подборка ссылок на примеры кода, подготовленная редакцией журнала Си User's Journal.

http://www.cuj.com/link/index1.html

Примеры кода

Примеры кода для Powersoft Power++.

http://www.orbit.org/power/code.html

Примеры кода

Примеры кода для Microsoft Visual C++.

http://msdn.microsoft.com/visualc/downloads/samples.asp

Ссылки для Visual C++

Сборник ссылок, посвященных Microsoft Visual C++.

http://www.utu.fi/~sisasa/oasis/oasis-windows.html

Примеры кода

Примеры кода, опубликованного в Windows Developer's Journal.

http://www.wdj.com/code/archive.html

Список использованной литературы

Справочное руководство по С++

Бъерн Страуструп, Библиотека М. Машкова

Сборник часто задаваемых вопросов и ответов к ним по языкам Си и С++

Arkady Belousov aka ark@munic.msk.su.

Сборник часто задаваемых вопросов и ответов к ним по WATCOM

Cyril Pertsev, Dmitry Kukushkin, Dmitry Turevsky, Alexey Dmitriev, Sergey Zubkov, Edik Lozover, Борисов Олег Николаевич, Alex Lapin, Alexandr Kanevskiy, Maxim Volkonovsky, Serge Romanowski, Serge Pashkov, Oleg Oleinick, Aleksey Lyutov, Anthon Troshin, Dmitry Ponomarev, Andy Chichak, Sergey Cremez, Vadim Belman, Vadim Gaponov, Владимир Сенков/Doctor Hangup, Roman Trunov, Jalil Gumerov, Anton Petrusevich, Kirill Joss, Lenik Terenin.

Лабораторные работы

Лицентов Д.Б.

Язык Си

М.Уэйт, С.Прата, Д.Мартин

Язык Турбо Си

Уинер Р.

Язык Си: введение для программистов

Берри Р., Микинз Б.

TURBO С++

Borland International Inc.

Реализация списка

Лицентов Д.Б.

Сборник часто задаваемых вопросов и ответов к ним по компиляторам языков Си и С++

http://soft.munic.msk.su/

Конференции relcom.fido.su.c-c++
Александр Кротов

Язык программирования Си
Шолмов Л.И.

Трюки программирования
http://www.kalinin.ru/programming/cpp/

Правило «право-лево»
Alexander V. Naumochkin

Effective C++
Scott Meyers

Advanced C++: Programming Styles and Idioms
James O. Coplien

Sutter's Mill: What's in a Class?
Herb Sutter

Exceptional C++
Herb Sutter

Large-Scale C++ Software Design
John Lakos

(B)leading Edge: How About Namespaces?
Jack Reeves

Personal communication
Jack Reeves

Эффективное использование C++. 50 рекомендаций по улучшению ваших программных проектов
Скотт Мейерс

Автоматизация и моторизация приложений
Николай Куртов

Обзор C/C++ компиляторов EMX и Watcom
Олег Дашевский

Рассуждения на тему «Умных» указателей
Евгений Моисеев

Использование директивы #import в Visual C++
Игорь Ткачёв

Язык Си в вопросах и ответах
Стив Саммит
Крупник А.Б. Перевод с английского.

Создание системных ловушек Windows на Borland C++
А.Е. Шевелёв

Четвертый BORLAND C++ и его окружение
М.Вахтеров, С.Орлов

Серия книг «Справочное руководство пользователя персонального компьютера»

Сабуров Сергей Викторович

Языки программирования C и C++

Налоговая льгота
«Общероссийский классификатор
ОК 005-93-ТОМ2 953000 — Книги и брошюры»

Лицензия ЛР № 068246 от 12.03.1999 г. Подписано в печать 19.02.2006. Формат 70x100/16.
Бумага газетная. Кол-во п.л. 41. Тираж 3000 экз.